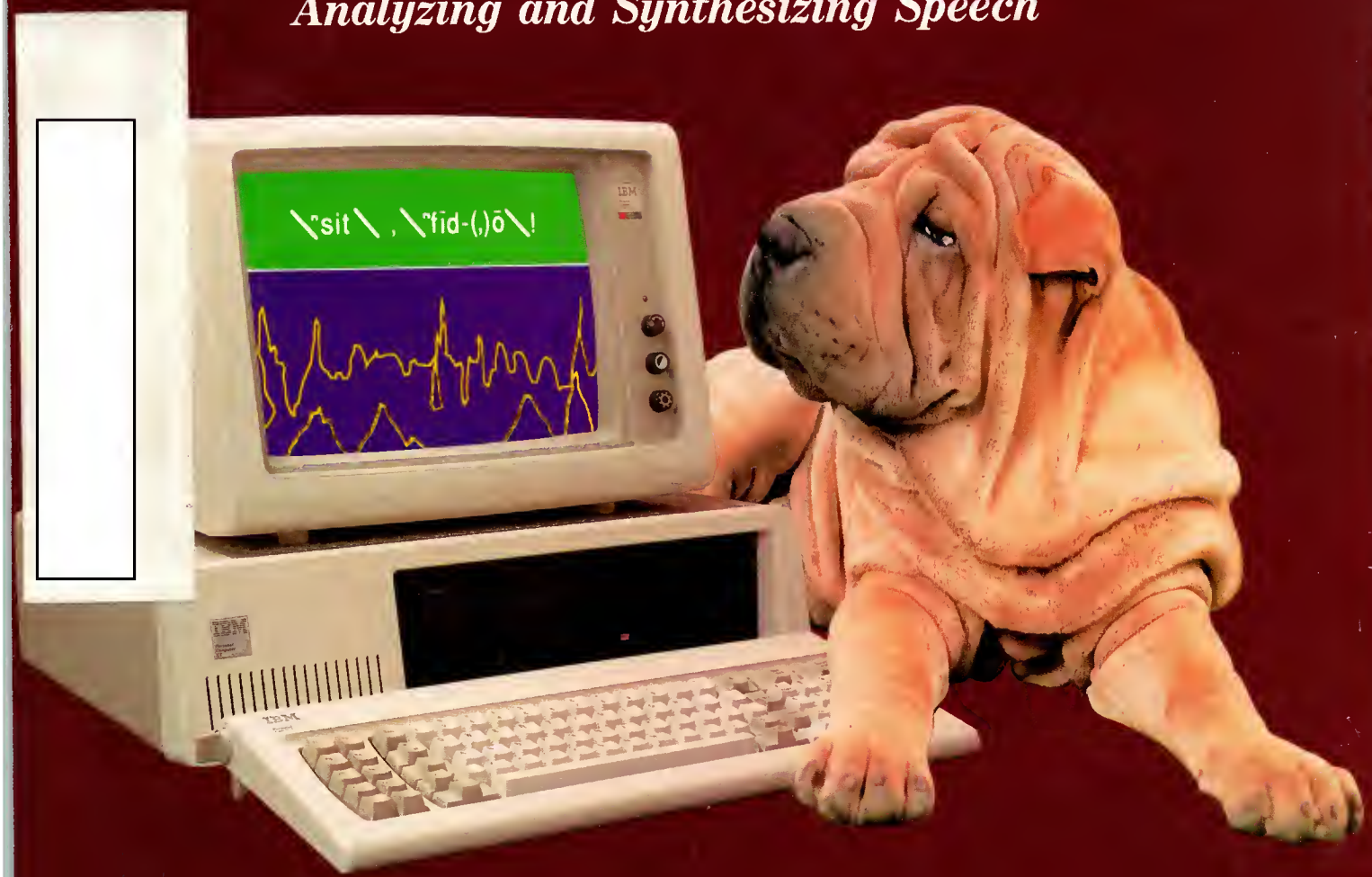
IEEE MICRO

JUNE 1987

Is he hearing a familiar voice?

Analyzing and Synthesizing Speech



Also in this issue:

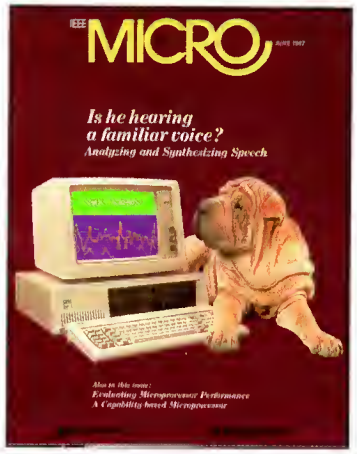
Evaluating Microprocessor Performance

A Capability-based Microprocessor

IEEE MICRO

Volume 7 Number 3 (ISSN 0272-1732) June 1987

Published by the Computer Society of the IEEE



On The Cover

Will scientists be able to synthesize human speech so clearly we'll be able to understand it as easily as we do the voices on our stereo records and tapes? They're trying and—interestingly enough—they're using PCs to analyze this complex task. Turn to p. 4 for more information.

Departments

Feature Articles

Circulation: IEEE MICRO (ISSN 0272-1732) is published bimonthly by the Computer Society of the IEEE: IEEE Headquarters, 345 East 47th St., New York, NY 10017; Computer Society West Coast Office, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Annual subscription: $17 in addition to Computer Society or any other IEEE society member dues; $25 for members of other technical organizations. This journal is also available in microfiche form.

Postmaster: Send address changes and undelivered copies to IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Second class postage is paid at New York, NY, and at additional mailing offices.

Editorial: Unless otherwise stated, bylined articles and descriptions of products and services in New Products, Product Summary, MicroReview, MicroNews, and MicroLaw, reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the Computer Society of the IEEE. Send editorial correspondence to IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All submissions are subject to editing for style, clarity, and space considerations.
IEEE MICRO subscribes to The Computer Press Association's code of professional ethics.

BPA

Q:

Involved with Bus/Board Problems and Products?

A:

The U.K. Bus/Board Users Show & Conference!

13/14 October, 1987
Excelsior Hotel, Heathrow, London

Conference Chairman: Dr. Paul Borrill, Spectra-Tek, Ltd.

- **CHARACTERISTICS:** A highly-concentrated technical forum with relevant tutorial exhibits for designers, developers, specifiers and systems integrators involved in bus architecture and board-level applications and usage.
- **TECHNICAL SPECIFICATIONS:** Technical sessions and seminars on subjects such as: backplanes • interfaces • real-time (Unix or Kernels) • system architecture, I/O applications, high-speed processing • tools for designers • shared memory • specific bus applications.

EXHIBIT PRODUCT CATEGORIES

Board Manufacturers • Systems Manufacturers • Packaging • Card Cages • Connectors • Surface Mount Devices • Software

BUS CATEGORIES

PC • Multibus I • Multibus II • Versabus • Q-bus • NuBus • VMEbus • Futurebus • STD Bus • S-100 Bus • G-64 & G-96 • Unibus • Cimbus • Exorbus • CAMAC • AMP • SMP • FASTBUS • STE Bus • I²C • BITBUS • C-44 • SCSI • Proprietary • *and more!*

For more information: Telephone or write: Roger Sherman, (Buscon U.K. Coordinator), Overseas Trade Show Agencies, Ltd., 11 Manchester Square, London W1M5AB • Phone: 01-487-2983 • Telex: 24591 Montex G • Fax (USA): 213 402 8814

BUSCON EUROPEAN BUSINESS TRAVEL PACKAGE AVAILABLE

Designed for U.S. Bus/Board manufacturers, this package will take you to London, Amsterdam and Munich, October 10 thru 22, 1987. The itinerary includes: • BUSCON UK Conference • SYSTEMS show • Private meetings in each city with government & industry leaders and the Press • Full staff support • Regularly scheduled British Airways flights • First Class hotels, breakfast, transfers, taxes • Customized tours.

This is a unique opportunity to travel with your peers to three important European business capitals. *There is no way you can duplicate this package on your own.*

Contact Anne Weber in California at 213-402-1610 for details, no later than July 10th.

From the Editor-in-Chief

SJ replies

"Thanks for the extra issue of *IEEE Micro*. I will actively encourage others to join the Computer Society." SJ, Austin, TX.

After receiving SJ's note, I spoke with him by telephone. He felt that his time constraints prohibited him from renewing *IEEE Micro* right now. However, he does continue to receive *Computer* and *IEEE Software*.

Mailbag

In addition to SJ's response, there were 34 cards in the mailbag. So far only four responses have been received on the April TRON issue. The remainder refer to February or earlier:

"Excellent issue...especially BTRON." R.W., Decatur, AL

"Best issue in the last three years (at least). The TRON project is something we all ought to know more about." J.L., Minneapolis, MN

"Too much space for TRON...wasted issue...Arno's (Peel) paper excellent and useful." D.T., Fairfax, VA

"Liked MicroLaw and New Products . . .wasn't interested in TRON." D.T., Lexington, KY

"I liked all the papers...." P.A., Khorasan, Iran

"...issue on multiprocessing was very good." G.S., Bombay, India

"Liked DSPs." C.M., Barton, Australia

"Liked the practical aspects of DSPs." A.B., Stevenage, UK

"DSP issue, excellent articles." E.L., Nedlands, Australia

"Liked 1987 editorial calendar." C.G., Buenos Aires, Argentina

"Liked DSP56000 and ADSP2100 articles." Ramallah, Israel

"Loved the letter from Fletcher J. Buckley!" B.S., Berkshire, UK

"(lengthy comment)...I would like (the usual): more articles, more often." I.S., Cambridge, MA

"I like the new MicroStandards, but let's have more on objectives and the status of standards." B.W., North Hollywood, CA

"MicroLaw very lucid." G.G., Port Angeles, WA (During my entire association with this magazine, no aspect has received more consistently favorable comments than MicroLaw. We are indeed fortunate to have a contributor like Dick Stern.—JF)

"Oops! Photos on page 88 are reversed." A.W. Lewisburg, PA (Yes, you are correct. You have sharp eyes. You are not alone.— JF)

"I liked the new format, more color, MicroNews, and parallel processors. Pictures on page 88 reversed." K.S., Acton, MA

"I liked MicroLaw and this issue." J.A., Fairfax, VA

"I liked the article on 'FFT Implementation Alternatives.'" H.D. Brampton, Canada

"I liked 'FFT Implementation Alternatives.'" S.G., Newcastle, UK

"I liked MicroStandards and Letters to the Editor." J.G., Fishkill, NY

"The FFT article outdated." Z.G., Belfast, UK

"I liked all of it (February issue)." A.B., Mexico City, Mexico

Final note: With this issue George S. Carson completes his terms on the *IEEE Micro* editorial board. Carson did an excellent job not only in reviewing articles but also in chairing our editorial board search committee. Our new assistant editor, Christine Miller, joins our staff in Los Alamitos this month. Her biography appears on page 88.

Best regards,

Jim Farrell

An IBM PC XT enhanced with speech boards and added memory permits researchers to experiment in language synthesis before final commitment to a target speech synthesizer.

F E A T U R E

A Personal Computer-based Speech Analysis and Synthesis System

Yousif A. El-Imam, IBM Kuwait Scientific Center

Speech analysis and reproduction is a popular theme in research today, involving phoneticists, computer engineers, and signal-processing and speech acoustics scientists. These studies attempt to understand the vocal sounds and patterns inherent in languages so that speech may be recreated synthetically, in as normal a fashion as possible.

Personal and mainframe computers equipped with a general-purpose signal processor allow researchers to segment, analyze, and synthesize speech for experiments before final commitment to a target synthesizer. Here we discuss a system centered on the IBM PC XT. The system, developed at the IBM Kuwait Scientific Center as part of a research project in speech synthesis, can be used in a stand-alone mode, or it can be enhanced by access to a mainframe computer.

Synthesizing a language from discrete units—such as short phonetic segments like allophones and demisyllables or long segments like words and phrases—requires facilities capable of carrying out several functions:

- digitization, quantization, and acquisition of speech signals;
- isolation of synthesis units from normal speech utterances;
- verification of the contextual variations occurring in the synthesis units during normal continuous speech;
- analysis and encoding (with suitable models) of the synthesis units; and
- development and use of an adequate synthesis strategy.

The use of a computer for processing a speech signal requires, first of all, that the signal be digitized, quantized, and acquired into computer memory. The isolation of synthesis units requires that the input utterance data be edited and segmented by some computerized facilities. Contextual variations must be verified by a phoneticist with perceptive judgment who can define each variation. The phoneticist, knowing the phonetic description of the language and the justification of the phoneticist view, is assisted by computer or other methods, such as the verification of the allophones of the basic phonemes and/or the study of speech prosody changes in pitch, stress, and rhythm. (A short glossary on the next page defines some of the specialized speech terminology.)

Speech analysis and encoding identifies the inherent temporal variables of the model used to represent the human speech production mechanism. (See the box on page 6 for general information on the speech process.) Speech must also be transformed into a coded, compressed, parametric form to save computer memory. One of two approaches, briefly described here, can be used to model the speech production process.

- The acoustical domain approach uses models whose parameters are measured from data of actual output speech. Model parameters are divided into source parameters (such as pitch, gain, and so on) and vocal tract parameters. These latter parameters reflect the time dependence of the spectral properties of the vocal tract and include variables such as the formants (resonance frequencies) of the vocal cavities [1,2] or linear predictive codes (LPC) of the discrete-time, time-varying model of the tract. [3-5]

0272-1732/87/0600-0004$01.00 © 1987 IEEE

Glossary of Speech Terms

An *allophone* is one of two or more variants of the same phoneme. (See phoneme below.)

A *demisyllable* is part of a syllable that consists of a consonant and part of a vowel.

A *diphone* is a phonetic segment that starts from the center of one phoneme and ends at the center of a neighboring phoneme.

A *formant* is any of several resonance bands held to determine the phonetic quality of a vowel or speech sound.

Fricative describes a consonant characterized by frictional passage of the expired breath through a narrowing at some point in the vocal tract.

The elongated space between the vocal cords is called the *glottis*.

Intonation is the rise and fall in pitch of the voice in speech.

Orthographics is the part of language study that concerns letters and spelling.

Phonemes are the smallest units of speech that serve to distinguish one utterance from another in a language or dialect; for example, the \p\ of *pat* and the \f\ of *fat* are distinctive in the English language.

Phonetics is the system of speech sounds of a language or group of languages as well as the study and systematic classification of the sounds made in spoken utterance.

Pitch is the difference in the relative vibration frequency of the human voice that contributes to the total meaning of speech.

Prosody is the study of versification, especially the systematic study of metrical structure.

The ordered recurrent alternation of strong and weak elements in the flow of sound and silence in speech is known as *rhythm*.

Stress describes the intensity of utterance given to a speech sound, syllable, or word producing loudness.

The membrane in the mouth resembling a veil or curtain is known as the *vellum*; it is also called the soft palate.

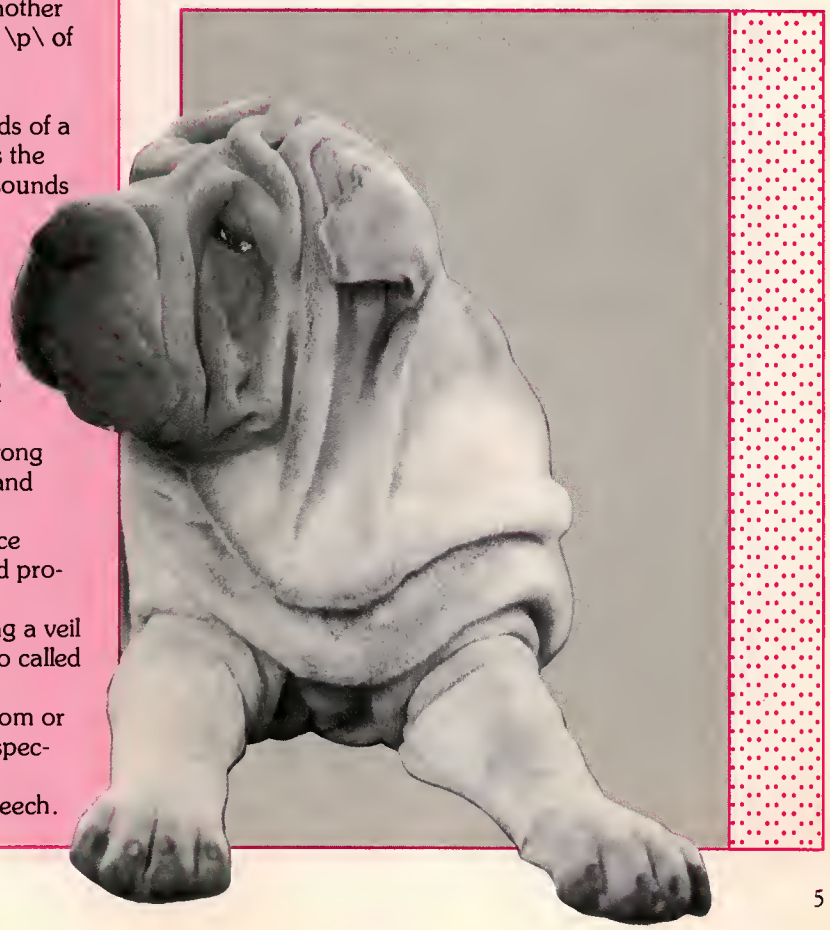
White noise is a term defining the random or impulsive noise that has a flat frequency spectrum at the frequency range of interest.

A speech *window* is a time frame of speech.

• The articulatory approach models the dynamics of the vocal tract directly, rather than modeling its acoustical output. This approach considers physiological parameters such as the positions of the tongue tip and the rounding of the lip. This approach to analyzing speech signals requires considerable computational effort in measuring the model parameters.[6]

Speech synthesis techniques use speech production models and a synthesis strategy (rules that use contextual knowledge about the synthesis units) to produce smooth parameter tracks for the target speech synthesizer. (The box on page 7 explains the methods of synthesizing speech.) With this data the synthesizer can produce continuous, intelligible, and possibly natural-sounding synthetic speech. The quality of the final synthetic speech depends on all the stages of the synthetic speech development process; neat speech editing and segmentation, accurate analysis and encoding, and complete strategy rules present better sounds.

Voluminous processing is required to measure and encode the speech model parameters. Often the number of synthesis units needed for a language can be very large. (Arabic, for example, requires on the



order of 1500 demisyllabic units using a syllabic approach for synthesis. In comparison, the English language requires on the order of a thousand synthesis units using a diphone approach.) Much repetitive editing and segmentation is, therefore, required. Often, implementing the synthesis strategy rules calls for very large and complex programs. Most of the time the complete process (from digitization and segment selection to synthesizing an utterance) must be repeated several times to produce a satisfactory synthesis.

Because of these considerations, good, reliable, versatile, and fast computer-based techniques become essential tools for developing the parameter codes to be used for speech synthesis.

Approaches and motivations. In the past synthetic speech was developed either by capable organizations in charge of speech development service bureaus or through the use of large speech analysis and synthesis packages that required the power of a minicomputer. A potential customer of the bureau would make speech recordings, then the bureau would develop and test speech parameters on a target synthesizer.

Few minicomputer-based speech research and development systems or general-purpose digital signal processing software programs have been developed.[7-9] Such systems and software are very powerful, but their use is limited to people with access to speech laboratories equipped with minicomputers.

In recent years with the development of microcomputer and microprocessor technology, many powerful microcomputers (in terms of processing speed and storage capabilities) have emerged. New support hardware (such as directly pluggable boards) allows easy attachment of the more unusual peripherals to personal computers. The personal computer is now very versatile in many new application areas.

Motivated by the advances in the technology of the personal computer and the microprocessor and the

The Speech Production Process and the Speech Waveform

The continuous movement of the articulators (tongue, lips, jaws, and velum) inside the vocal tract produces the sounds that create speech. The vocal tract is formed of two cavities, the oral and the nasal.

- The oral cavity extends from the glottis to the lips. During the production of speech sounds, this cavity also forms a nonuniform area that depends on the positions of the articulators.
- The nasal cavity extends to the nostrils and is formed by lowering the velum. The nasal cavity can become acoustically coupled to the oral cavity to produce the nasal sounds of speech.

The vocal tract is excited by the action of the lung muscles forcing air through two small muscular flaps at the larynx called the vocal cords. During the production of voiced sounds, the vocal cords vibrate to modulate the air from the lungs, producing quasiperiodic pulses of air. The period is the pitch, and the frequency of vibration of the vocal cords is the fundamental frequency of the sound produced.

Fourier analysis of the quasiperiodic pulses of air shows a discrete harmonic frequency structure of decaying amplitudes. During the production of voiceless sounds, the vocal cords are at rest and the excitation source is moved (from the larynx) to the point of constriction along the oral cavity. The constriction produces turbulent air flow and the excitation, thus produced, is measured as a broadband noise. During the production of plosive sounds, a complete closure forms somewhere along the oral cavity, allowing air pressure to build behind it. The suddenly released air pressure results in a very short burst of fricative noise.

No matter which excitation is used, the frequency spectrum of the output speech waveform is shaped by the frequency selectivity of the vocal tract. The vocal tract resonates at certain frequencies called the formants. The number and values of the formants depend on the area function (cross-sectional area as a function of distance along the tract and time) of the vocal tract.

In linear system theory, the vocal tract can be viewed as a time-varying linear system (or filter[24]) whose parameters are assumed constant over a short analysis period. During voiced sound production, the filter is excited by periodic pulses; during unvoiced sound production, the filter is excited by white noise. The filtering action of the vocal tract produces an output speech waveform of very complex nature. Frequency domain methods are, therefore, natural means for analyzing such a signal.

Figure A depicts the speech production process. Figure B shows a speech signal of an Arabic vowel (voiced sound segment) and Figure C, a frequency response of the oral cavity during the production of the vowel.

Speech Synthesis Methods

Speech can be synthesized by either of two methods: synthesis by analysis (also called analysis/synthesis) or constructive synthesis.[22] Both methods use a set of synthesis units from which synthetic speech can be generated.

In the synthesis-by-analysis approach, the synthesis units are long segments of speech such as words, phrases, or even sentences. The synthesis-by-analysis system encodes the acoustical representation of the units to achieve varying degrees of speech data compression:

- a set of parameters obtained under a technique called linear predictive coding and
- direct waveform coding such as the Mozer method[22] and the adaptive delta modulation[23] techniques.

Synthesis-by-analysis methods are characterized by the ability to generate naturally sounding speech, but they are expensive because they must store many synthesis units.

Constructive-synthesis methods use synthesis units that are discrete phonetic sound segments such as allophones, diphones, demisyllables, etc. Every human language has its own set of such sounds. The constructive-synthesis system must be capable of creating an inventory of such sound segments, suitably encoding them, and generating synthetic speech. These systems are characterized by their ability to generate an unlimited vocabulary of synthetic speech with less storage requirements than systems based on the synthesis-by-analysis method. However, the quality of the synthetic speech is not as good.

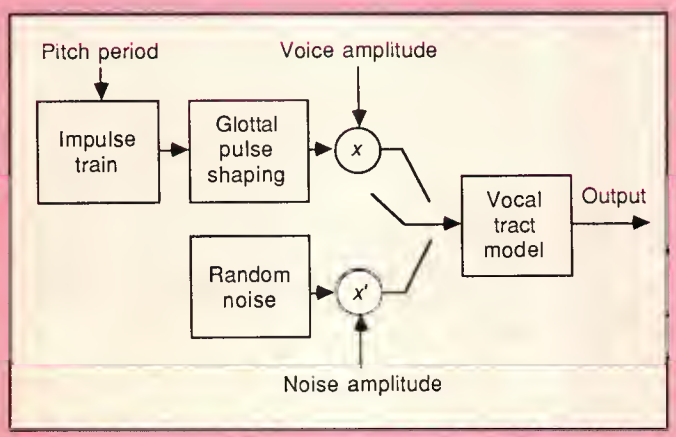


Figure A. The speech production model.

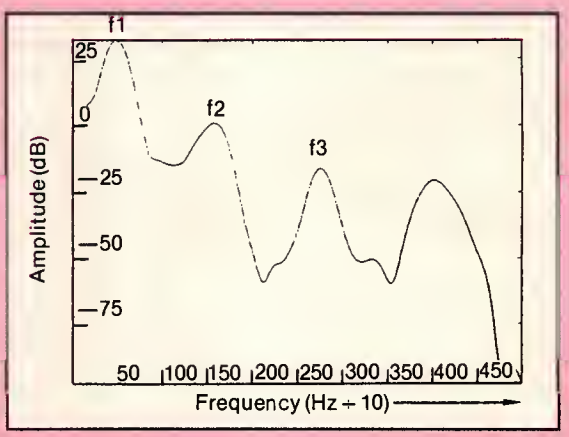


Figure C. Frequency response of the oral cavity during vowel production.

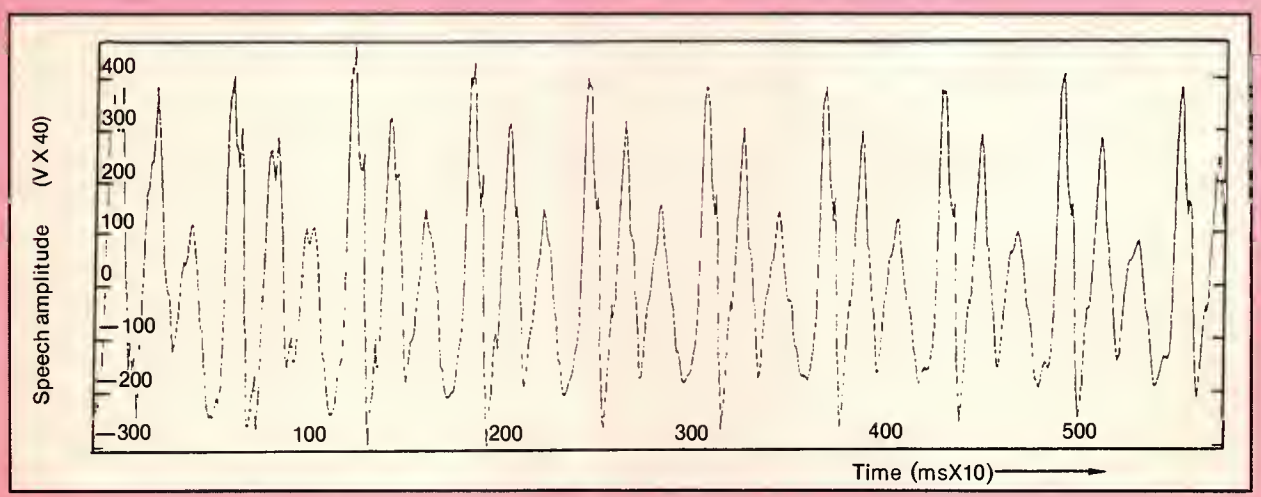


Figure B. Speech waveform of a long Arabic vowel.

need for simplifying and cutting down the cost and effort that goes into developing synthetic speech, some organizations have chosen to develop complete personal computer-based speech development systems (hardware and software).[10] Others have chosen to develop complete microprocessor-based speech development stations.[11] Still others have chosen to adapt already existing general-purpose signal processing packages to certain brands of personal computers.[12]

Here we describe a PC-based speech development and research system and compare it, as far as possible, to similar systems reported in the literature. The system can be used in a stand-alone mode or linked to a host computer in three domains of applications involving speech processing:

- An experimental research system for synthesizing a language from short or long phonetic segments such as allophones, diphones, demisyllables, words, and short phrases. For this purpose the system provides all the needed functions, from digitizing the speech to generating synthetic speech by a simulated LPC synthesizer.
- A tool for conducting language-dependent studies such as prosodic features and the verification of the contextual variations in the synthesis units. (Verification of the allophones of the Arabic language is an example.)
- A development tool for generating parameter codes (LPC or formants) that eventually could be adapted to a specific target synthesizer.

System configuration

As can be seen in Figure 1, the system we describe is based on an IBM PC XT computer with options and accessories: a 256K user RAM; a 360K floppy diskette drive; an IBM France Scientific Center speech board (FSCB);[13,14] a Techmar Labmaster board;[15] an IBM graphics adapter/display; an IBM expansion unit with two optional 10M-byte hard disks; and an optional IBM 3278 emulation board.

The FSCB board processes the input speech signal. A tenth-order, elliptical, switched-capacitor, low-pass filter initially filters the (analog) speech. The cutoff frequency of the filter varies, but in this implementation we keep it set at 4 kHz. A 12-bit analog-to-digital converter operating at a 10-kHz sample rate digitizes the filtered speech. The FSCB board computes—on a window basis (one window equals 12.8 ms of speech or 128 speech samples) in real time—several short-time speech parameters such as energy, zero-crossing rate, and pitch. (See the accompanying box for a description of the windowing process.)

The Tecmar Labmaster Board performs the final processing of the output speech signal. A 12-bit digital-to-analog converter operating at a 10-kHz sample rate changes the analog signal to digitized speech. The signal is filtered by a sixth-order, elliptical, switched-capacitor, low-pass filter. As on the FSCB board, the filter cutoff frequency is variable but set to 4 kHz here. The output of the filter is coupled through a 100k-ohms audiotape device to the audio output subsystem (preamplifier, power amplifier, cassette recorder, and speakers). A Model 3278.3 graphics display screen supplements the IBM 4341/2 mainframe host computer. An IBM 3278 emulation board and an emulation program connect the PC XT and the host.

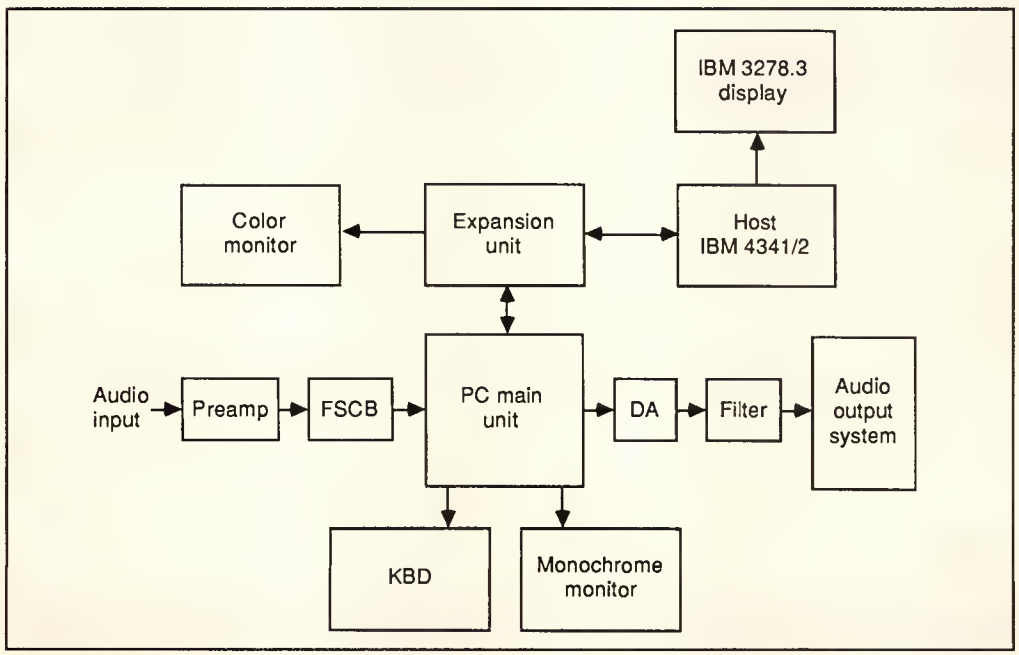


Figure 1. System configuration.

Short-time Analysis and Speech Windowing

By the nature of the speech production process, the speech signal can be viewed as a signal whose properties vary with time. The speech production process was modeled (in the box on page 7) by a time-varying, discrete-time system. The parameters of such a model can be assumed to be constant over a short-time analysis interval. This constancy makes short-time analysis a natural way to measure the time dependency of the speech production process.

There are two types of short-time parameters: *time-domain* parameters such as the short-time energy, pitch, and zero-crossing rates; and the *autocorrelation function* and *frequency domain* parameters such as the short-time Fourier transform and the LPC coefficients.

To be able to conduct short-time analysis on the speech signal, we must see that the signal is windowed by an appropriate window of appropriate length. Mathematically, the windowing process can be viewed as a convolution of a transformed version of the speech signal with the window as shown by the following equation:

$$Q_n = \sum_{m=-\infty}^{\infty} T[X(m)]w(n-m) \qquad \text{(A)}$$

where $T[x(m)]$ is a linear or nonlinear transformation of the sequence $x(m)$, and $w(n)$ is a finite-duration window positioned at time index n.

Two types of windows are in common use for short-time analysis: a Hamming window, whose impulse response is given by

$$\begin{aligned} h(n) &= 0.54 - 0.46\cos[2\pi n/(N-1)], \quad 0 \le n \le N-1 \\ &= 0 \qquad\qquad\qquad\qquad \text{otherwise} \end{aligned} \qquad \text{(B)}$$

and a rectangular window of impulse response given by

$$\begin{aligned} h(n) &= 1 \qquad 0 \le n \le N-1 \\ &= 0 \qquad \text{otherwise} \end{aligned} \qquad \text{(C)}$$

where N is the window length in speech samples.

Short-time windowing has four important functions:

- it emphasizes the part of the speech signal to undergo analysis and sets the signal to zero outside the analysis frame;
- when properly chosen, it gives a clear indication of the time-dependent properties of the speech signal;
- it represents a smoothed version of the spectral properties of the signal inside the window; thus, if the spectral properties of the signal are uniform outside the analysis frame (a sustained speech segment), a short-time Fourier transform, for example, should represent the average properties of the signal outside the analysis frame; and
- in some cases, it guarantees the existence of the short-time quantity being measured.

Rabiner and Schafer give a more complete discussion of short-time analysis and the effect of windowing on the various short-time parameters.[16]

The type of window used has an important effect on the properties of the short-time quantity being measured. We illustrate this by considering the short-time Fourier transform. A useful definition of this transform appears in the following form:

$$X_n(e^{jw}) = \sum_{m=-\infty}^{\infty} w(n-m)\,X(m)e^{-jwm} \qquad \text{(D)}$$

where $w(n)$ is a window (Hamming or rectangular) and $x(m)$ is the speech signal. We can interpret this equation in two ways:

- as the normal Fourier transform of the sequence $w(n - m)x(m)$; or
- as the convolution of the window $w(n)$ with the quantity $x(m)e^{-jwm}$.

The first interpretation leads to the fact that, for the normal Fourier transform of the sequence to exist, the condition

$$\sum_{n=-\infty}^{\infty} | w(n-m)x(n) | < \infty \qquad \text{(E)}$$

must be satisfied. This is true since the window $w(n)$ has a finite duration. This explanation shows how windowing can help guarantee the existence of certain short-time parameters.

The second interpretation leads to better insight into the characteristics of the window $w(n)$ in the frequency domain and in terms of linear filtering theory. Ideally, we want the window impulse response to approximate a low-pass filter of cutoff w. The sharper and smaller the cutoff frequency (narrow-band analysis) is, the better the frequency resolution of the window; the larger the cutoff

(wide-band analysis) is, the better the time resolution of the window.

Figure D presents impulse responses of the rectangular and Hamming windows. As seen, these responses hardly approximate an ideal low-pass filter. They are characterized by a main lobe of given bandwidth and side lobes of certain levels. For both windows the bandwidth is inversely proportional to the window length, and for a given length it is narrower for a rectangular window than for a Hamming window. The levels of the side lobes are independent of the window length, but they are higher for the rectangular window than they are for the Hamming window.

Thus, in general, if speech analysts are interested in recovering the periodicity (harmonic structure) from short-time Fourier transform analysis, they should use a window of longer length. A rectangular window shows harmonic structure better than does a Hamming window; however, because of the large level of side lobes, the short-time Fourier transform of the rectangular window is noisier than the Hamming window. As a result, analysts seldom use rectangular windows in short-time spectral analysis of the speech signal.

On the other hand, the smaller the window length is, the poorer will be the frequency resolution (because of the wide bandwidth of the window). But the smaller window length will average the complete spectral properties of the speech signal better; it will also represent the short-time Fourier transform better to such temporal variables of the signal as the formants.

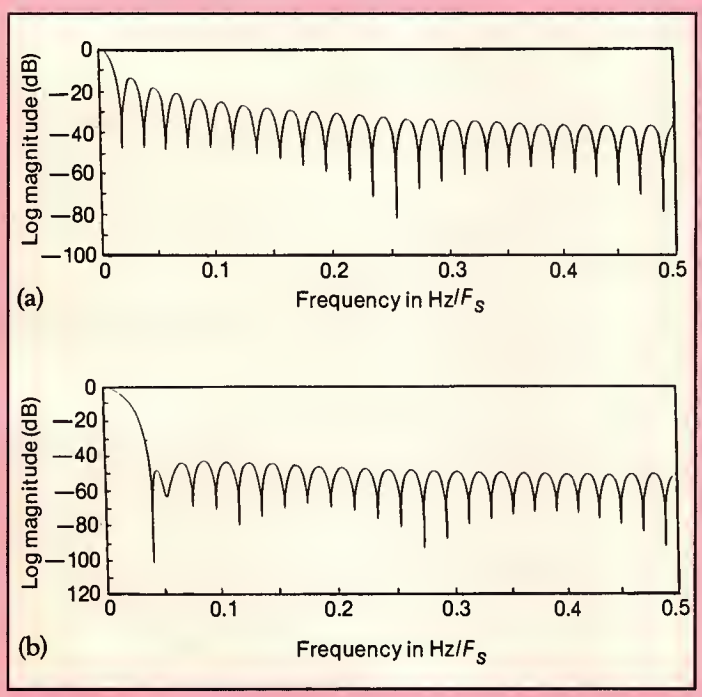


Figure D. Impulse response of a rectangular window (a); impulse response of a Hamming window (b).[16] (© 1978, Prentice-Hall, Englewood Cliffs, New Jersey. Reprinted with permission.)

System functions

The PC XT exercises control over all aspects of the system by providing, through the use of application software and a system menu, the following interactive functions:

- speech editing and segmentation,
- speech analysis and encoding,
- speech synthesis,
- speech prosodic analysis, and
- connection to the host.

Figure 2 summarizes the work carried out by each system function and shows the interaction between the menu program and the programs implementing those functions. Users select menu items with the help of function keys as shown in the figure. The menu program is reentered once a program implementing any specific function is completed. The following sections detail the achievements of these functions and the programs implementing them.

Speech editing and segmentation. Computer-based speech editing allows the speech scientist to input speech, select the segments of speech that are of interest to his application, and store the selected segments in a backup medium for any future use. With speech synthesis, the number of synthesis units can be very large, and much repetitive editing may, therefore, be necessary. This requirement necessitates efficient and fast interactive computer-based editing and segmentation facilities.

The present interactive speech editor offers user prompts to aid in beginning any required action. The segmentation of the speech data is not carried out directly on waveform displays. The editor program uses the FSCB board capability of evaluating the short-time energy and zero-crossing rate in real time. From the FSCB board, the program retrieves window values for the energy and the zero-crossing rate (a maximum of 256 window values, equivalent to about 3.3 seconds of speech) and displays them. Users can then move an interactive graphics cursor to select and isolate any segment of speech of interest to them.

The use of the short-time energy and zero-crossing rate for speech segmentation is well known and has certain advantages.[16] First, users can easily select the speech segment of interest because the classification of speech segments into voiced and unvoiced forms is easily seen from energy and zero-crossing displays. Second, because the speech is displayed in compressed form (only 256 points are used to represent a speech waveform of 65,536 samples), users can work with a low-resolution graphics terminal instead of the usual high-resolution terminals. In this implementation we use the medium-resolution (320×200) IBM graphics terminal in the color mode.

The editor program selects any segment from acquired speech data in two stages. In the first stage

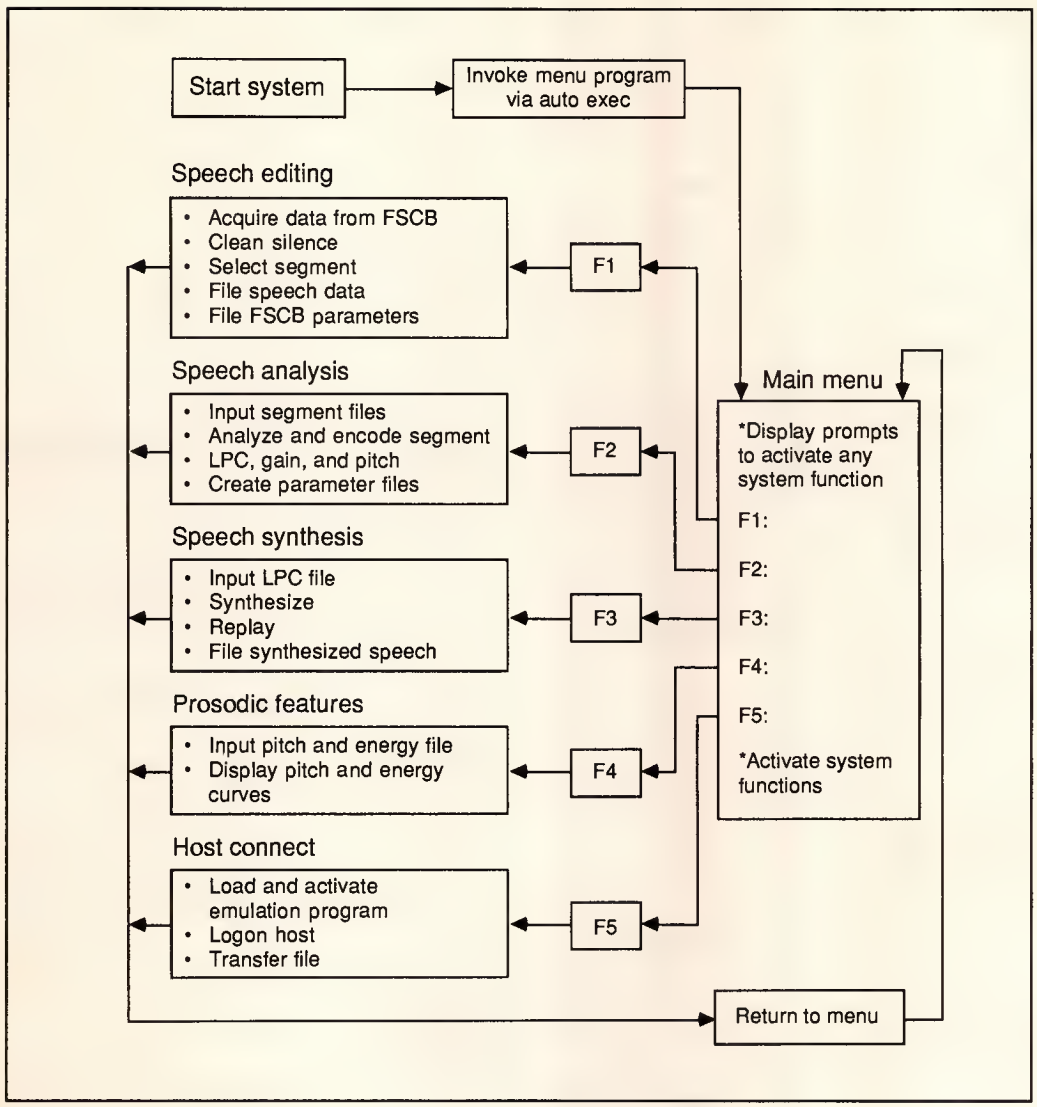


Figure 2. A summary of system functions and menu function selection.

users eliminate (from energy display) any silence in the acquired speech. In the second stage users select (from energy and zero-crossing-rate displays of the cleaned speech) the speech segment of interest. After this, users can display and view selected speech samples.

Other functions carried out by the editor are:

- repetitive capture and replay of any utterance until satisfactory hearing is achieved,
- optional filing (on the PC hard disk) of the cleaned speech and of any selected speech segment,
- retrieval (from the hard disk) of any previously filed, cleaned speech for any further segmentation, and
- spectrum evaluation and display of any portion of a voiced speech segment.

The editor allows users to speedily create on the PC XT hard disk an inventory of speech synthesis units (short phonetic segments like the demisyllables or complete word utterances). These units could be used later in speech synthesis research or application. Segment duration measurements are readily available from the editor program. These measurements are useful in language-dependent studies such as the verification of the contextual variations of the synthesis units and the study of some prosodic features such as changes in speech rhythm.

Additionally, the editor allows two speech waveforms to be active simultaneously in the PC RAM. Users can select and swap segments of equal lengths between the two waveforms and alternately replay them. We found that this facility can be of some use in the study of certain linguistic phenomenon such as the source of emphasis in the Arabic language.[17]

The editor program is a hybrid between several assembly language routines and a compiled Basic main program. Compiled Basic is used for the main

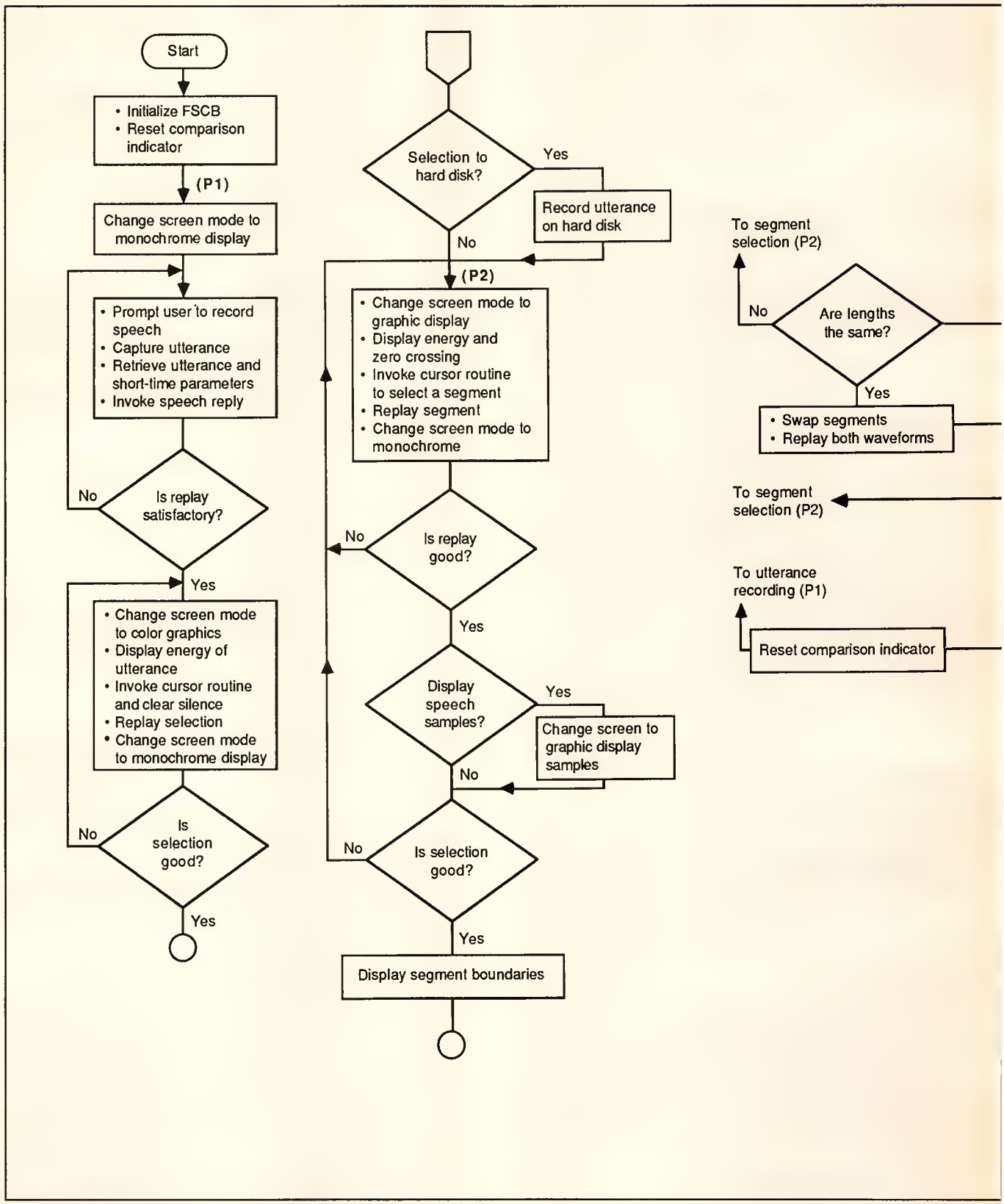
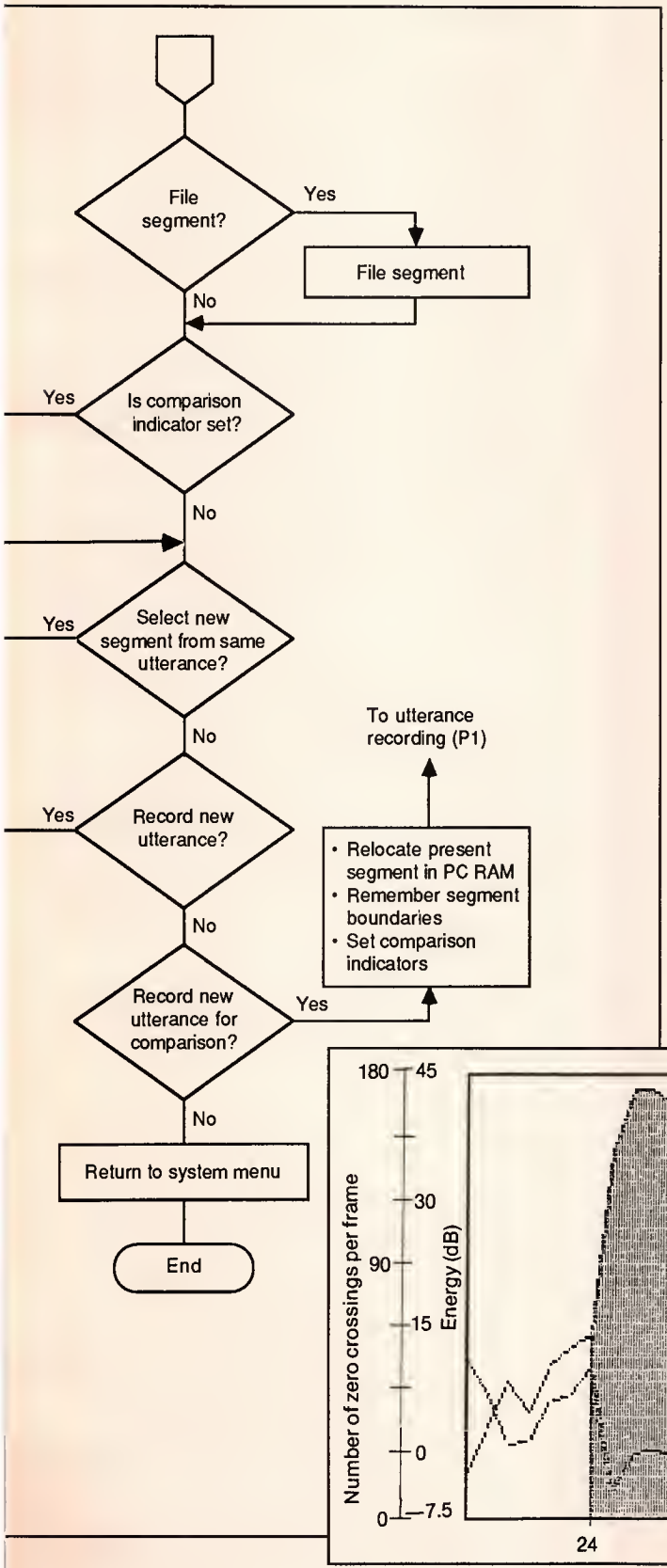


Figure 3. Flowchart for the speech segmentation program.



program because it interfaces easily to the FSCB board and the assembly language routines, runs as fast as any other compiled language for the PC, and has superb color graphics capabilities. Assembly language routines carry out low-level hardware-dependent tasks such as:

- controlling the digital-to-analog conversion and replay of the speech,
- relocating the speech from the FSCB into the user-free RAM, and
- switching the display mode between monochrome text and color graphics.

The editor uses both monochrome and color graphics screens. The monochrome screen outputs the text of user prompts and editor response messages, while the graphics screen is used exclusively for graphic output. This mode of operation is less confusing for users than using one text/graphics screen for simultaneous text and graphics output.

For any segment or utterance being processed, the editor outputs two disk files: a file of raw speech data for the segment or utterance being processed and a file of the short-time parameters for the same utterance. The flowchart in Figure 3 represents the way the program implements the speech-editing functions. Figure 4 shows the way a segment of speech is selected from short-time energy and zero-crossing curves.

Speech analysis and encoding. Analyzing and encoding speech as LPC parameters, pitch, and energy-related gain is accomplished through short-time

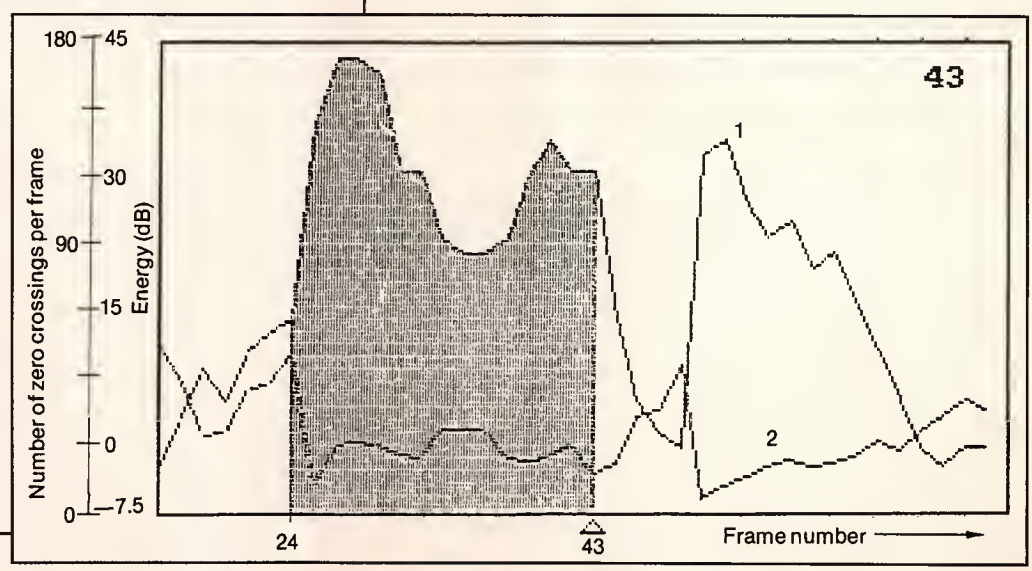


Figure 4. Interactive selection of a speech segment; energy (1), zero-crossing rate (2).

analysis techniques. (See the box on LPC techniques and computations.) The interactive speech analysis and encoding program takes, as input, the two files created by the editor and carries out, on a window basis, the following computations and functions:

- 14 autocorrelation coefficients from which 14 LPC parameters are derived,
- 14 LPC parameters,
- pitch and gain,
- optional display of window samples and the window autocorrelation function,
- optional graphical representation of the LPC parameters,
- optional display and editing of the LPC parameters, and
- filing of the parameters.

The display of window samples and the autocorrelation function allows users to make an empirical judgment about whether the window is voiced or unvoiced. Based on this decision, the program either computes or does not compute a pitch for the window. This step is important since some speech-voiced fricative windows—the shape of the autocorrelation function—while showing voicing presence do not really exhibit the sharp peaks necessary to satisfy the automatic pitch-detection criterion described in the box.

The graphical representation of the LPC parameters permits users to make spectral comparisons between segments of the same class. This type of comparison is useful in conducting language-dependent studies such as the verification of a language's allophones.

One may wonder why the process of finding the pitch is followed when the FSCB readily evaluates it in real time. The reason is that the FSCB pitch computation is tailored toward evaluating the pitch of sustained, uniform-voiced speech sounds. (For example, children with pitch irregularities could learn to control the pitch of their voices by using computer-based pitch-activated games.[14]) For this purpose the FSCB is very reliable; however, it is not suitable for normal continuous speech.

In synthesizing speech, it is always better to extract the synthesis units from continuous speech. The least this pitch computation will do is to verify whether the calculated pitch agrees with the FSCB board's pitch or not. If the results are not the same, users can depend on their own phonetic knowledge about the speech segment being encoded to decide whether to give the disputed window a pitch value or zero pitch—guided also by neighboring values.

The speech encoding and analysis process progresses interactively, one window at a time. The parameter values for each window in an utterance are computed, verified, edited (if necessary), and added to a parameter disk file. The compiled Basic program

LPC Technique and Computations Performed by the Analysis Program

Linear predictive coding is one of the well-known frequency domain techniques for speech analysis and synthesis. With this technique the vocal tract is modeled by a time-varying digital filter whose parameters are assumed constant over a short-time analysis window. The filter is either excited by periodic pulses for (voiced speech sounds) or by random noise (for unvoiced sounds).

A well-known property of linear predictive coefficients is their relationship to the geometry of a concatenation of lossless acoustic tubes. This property gives a physical correspondence between the LPC model and the speech production process discussed earlier. The vocal tract model used in the following computations is a 14-pole, time-varying digital filter whose steady-state system function is of the form:

$$H(z) = \frac{G}{1 - \sum_{i=1}^{14} a_i z^{-i}} \qquad \text{(a)}$$

The parameters that we need to measure from real speech data are the gain (G), the 14 LPC coefficients (α), and the pitch of the speech (for voiced sounds). The algorithms behind the computations of these parameters are well explained in the literature. The speech analysis program implements the algorithms. We give a brief overview here.

LPC computations

The first set of computations finds the 14 autocorrelation coefficients from which the vocal tract model parameters are derived. The 14 autocorrelation coefficients satisfy the relation

$$R(k) = \sum_{i,k} y(i) * y(i+k) \qquad 0 \le i \le 127 - k, \quad 0 \le k \le 13 \qquad \text{(b)}$$

We obtain the sequence $y(i)$ by multiplying the original window data by a Hamming window of equal length and of the form

$$H(i) = 0.54 - 0.46 * \cos(2 * \pi * i/127) \qquad 0 \le i \le 127 \qquad \text{(c)}$$

The computed set of 14 autocorrelation coefficients represent the window's low-time autocorrelation values. They also represent the frequency spectral characteristics of the window and are thus used to compute the 14 linear predictive coefficients.

We use the autocorrelation method[3] to obtain the linear predictive coefficients. Durbin's recursive

algorithm provides the predictor estimates.[25] The following recursive equations summarize this algorithm:

$$e^{(o)} = R(o) \tag{d}$$

$$k_i = \left[R(i) - \sum_{j=1}^{i-1} \alpha_j^{(i-1)} R(i-j) \right] / e(i-1) \qquad 1 \le i \le 14 \tag{e}$$

$$a_i^{(i)} = k_i \tag{f}$$

$$\alpha_j^{(i)} = \alpha_j^{(i-1)} - k_i \alpha_{i-j}^{(i-1)} \tag{g}$$

$$e^{(i)} = (1-k_i)^2 e^{(i-1)} \qquad 1 \le j \le i-1 \tag{h}$$

Equations (d) through (h) are solved recursively for i = 1,2,3,...,14. The final solution for the LPC model coefficients is given by

$$\alpha_j = \alpha_j^{(14)} \qquad i \le j \le 14 \tag{i}$$

In the solution for the 14 LPC coefficients, the autocorrelation coefficients $R(k)$ are replaced by their normalized counterparts:

$$r(k) = R(k)/R(o) \tag{j}$$

This equation does not change the solution to the LPC model parameters but leads to the partial correlation coefficients, or PARCOR coefficients, with (k_i) satisfying the condition

$$-1 \le k \le 1 \tag{k}$$

This condition on the quantities k_i is the necessary and sufficient condition for the LPC vocal tract model to be stable.[4,5] The autocorrelation coefficients are computed with sufficient numerical accuracy so no instability problems are met as a result of rounding-off errors in computation.

Pitch computation

The second set of computations carried out on the window data deals with detecting the fundamental frequency of the window. The technique used for extracting the pitch of the window is essentially a variation of Sohndhi's method[26] for spectrum flattering to remove the effects of the vocal tract transfer function on the window autocorrelation function. This method leads to peaks, (produced essentially because of the voice pitch), which show on the autocorrelation function. We summarize the technique:

- Send the original window data through a 900-Hz, digital low-pass filter having a discrete transfer function of

$$D(z) = \frac{z^{-2} + a_1 z^{-1} + a_2}{z^{-2} - b_1 z^{-1} + b_2} \qquad a_1 = 2;\ a_2 = 1;\ b_1 = 0.845;\ b_2 = 0.25 \tag{l}$$

- Compute the maximums in the first and last thirds of the filtered output, and the signal is clipped to 68 percent of the minimum of the two maximums.
- Use the output from the clipper to create a set of 128 autocorrelation coefficients (window autocorrelation function), which are used for tracking the pitch. Compute the autocorrelation function from the relation

$$R(p) = \sum_{i,p} z(i) * z(i+p) \qquad o \le i \le 127 - p;\ o \le p \le 127 \tag{m}$$

- A window is automatically indicated as voiced if the peak in the autocorrelation function is 30 percent of its value at zero time. Figure E shows the autocorrelation function and the speech samples of a voiced window.

The window gain is computed from the following relationship:

$$G^2 = R(o) - \sum_{k=1}^{14} \alpha_k R(k) \tag{n}$$

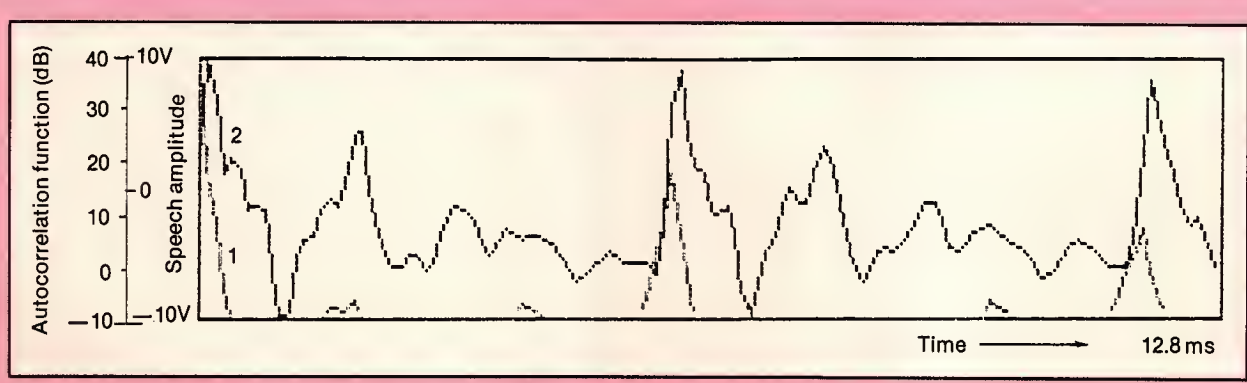


Figure E. Autocorrelation function (1) and speech samples (2) for a voiced window.

uses the assembly language routine to switch between monochrome and color graphics screens. The flowchart in Figure 5 shows the logic of the program.

Speech synthesis. A speech synthesis program, which is a software simulation of an LPC synthesizer, takes as input the encoded parameter disk files and delivers synthetic speech through either a synthesis-by-analysis or a constructive-synthesis method. In constructive synthesis the program smooths out the transitions across phonetic segment boundaries.

The time-varying, discrete-time LPC model, whose function in the steady-state system is given by Equation a (in the box on LPC techniques and computa-

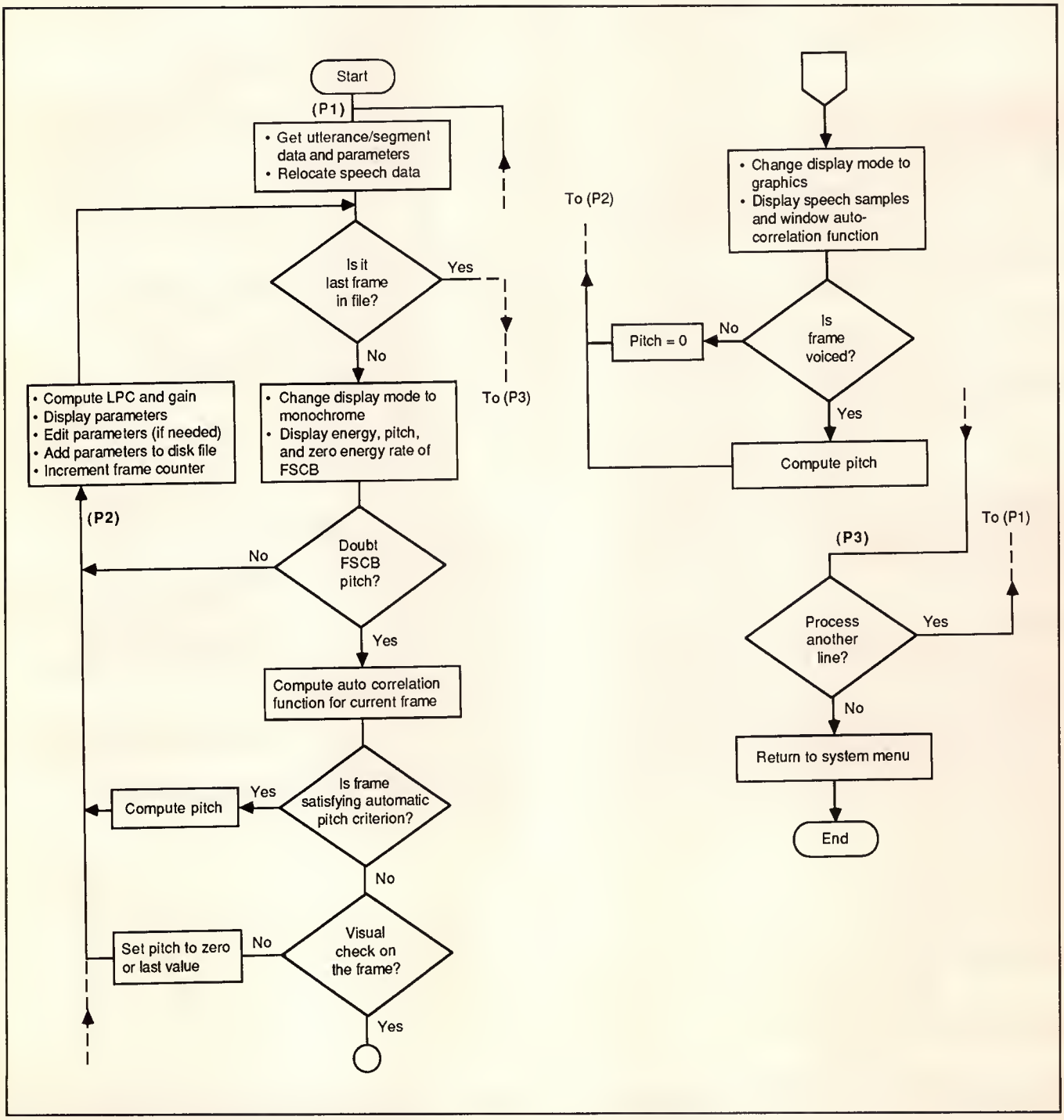


Figure 5. Flowchart of the speech analysis program.

tions), produces synthetic speech. For this system we relate the output speech samples $x(n)$ to the input excitation $u(n)$ by the following difference equation, which is of the recursive type:

$$x(n) = \sum_{k=1}^{14} a_k\, x(n-k) + Gu(n) \qquad (1)$$

The model is excited by an impulse train spaced at the fundamental period for a voiced speech window and by a wide-band random-noise sequence for an unvoiced window.

The quality of the synthetic speech is very good when synthesizing words and short phrases (for Arabic and other languages) with the synthesis-by-analysis method. The quality is also very good for synthesizing Arabic using a constructive-synthesis approach on demisyllabic units. However, for the time being, barely intelligible Arabic is produced using basic sound units such as allophones. Besides synthesizing and replaying speech, the speech synthesis program creates a disk file that holds the samples of the synthesized utterance. Users can also choose to view speech waveforms of synthetic and actual utterances.

Like the speech editor and the analysis programs, the synthesis program consists of a few assembly language routines linked to a main program. The main program is written in compiled Basic, and the same assembly language routines used with the editor are used here for the same purpose. The flowchart in Figure 6 shows the logic of the program; Figure 7 reproduces synthetic and actual speech waveforms for a voiced window.

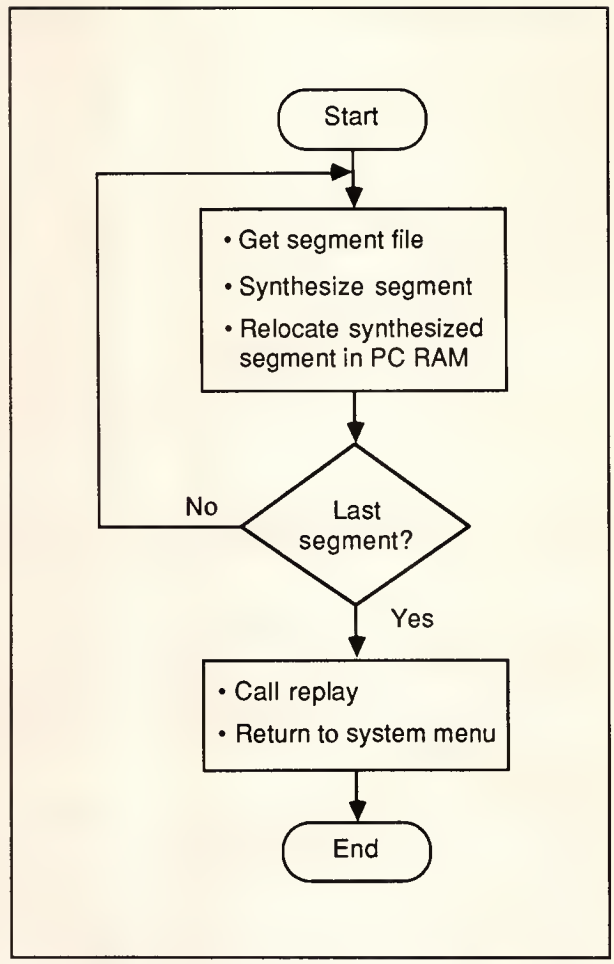


Figure 6. Flowchart of the speech synthesis program.

Prosodic features. The study of speech prosody in terms of variations that occur to the speech supra-segmental parameters (stress, pitch, or intonation and rhythm) during normal continuous speech is important in producing high-quality synthetic speech.

Such a study is needed for developing the rules underlying those variations and is important for constructive synthesis (synthesis by rule or text into speech). The rules, so developed, can be implemented by a text-to-sound transcription program, which analyzes input text and delivers smoothed parameter tracks for a synthesizer.

A program that plots pitch and energy contours as functions of time for a short sentence is included. The values of the pitch and energy for the sentence are obtained during the editing and the analysis

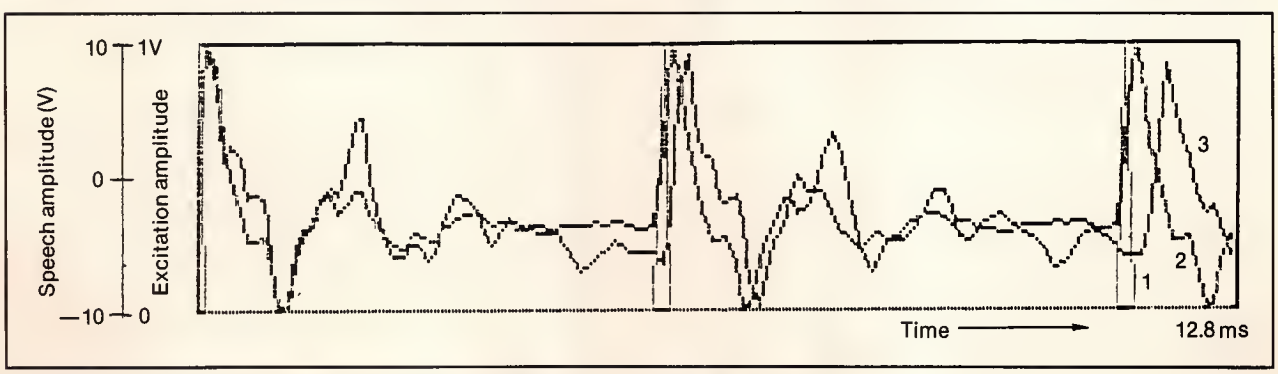


Figure 7. Examples of windows: excitation (1), synthetic (2), and actual voiced (3).

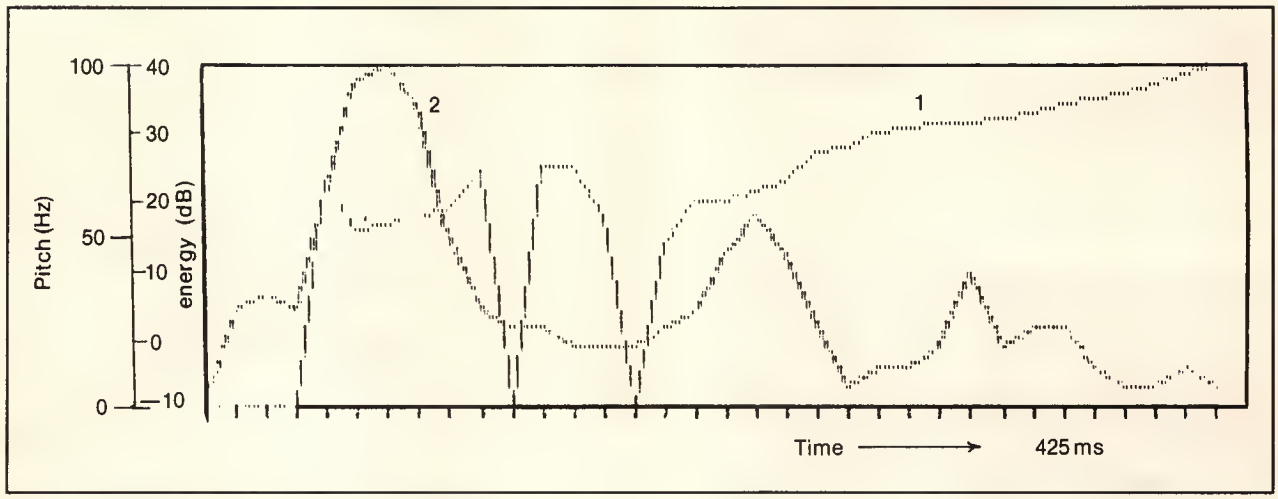


Figure 8. Pitch (1) and energy (2) contours for an Arabic word.

phases. Figure 8 shows such pitch and energy contours for a word made up of three syllables. From the shapes of these contours, it is obvious that the speaker has stressed the first syllable of the word and has said the word in an inquisitive context (indicated by the rising pitch).

Connection to the host and host analysis program. A high-speed link exists between the PC and the host computer. This link allows high-speed speech data transfer between the two computers. The connection to the host is useful as it allows users to take advantage of some powerful speech signal processing facilities available only at the host.

A powerful one- and two-dimensional signal processing package has been developed by the IBM Winchester Scientific Center.[18] This package can readily be used for speech processing applications. User-written macros in the IAX language (the language of the Winchester package) can perform the following speech-processing functions:

- spectrum analysis and display for any speech segment to extract such variables as the formants of certain voiced segments,
- spectrographic analysis and display for any utterance to show sound segment length and the formant movement in continuous speech,[19]
- cepstrum analysis for any speech segment to evaluate speech variables such as the pitch or the formants.[20,21]

The above analyses are useful if users are trying to encode speech in terms of formants (values, bandwidths, and amplitudes). They can also be useful if users are conducting language-dependent studies such as the contextual variations of the synthesis units. We have already used this host facility together with LPC analysis in conducting spectral studies on the Arabic language to verify its allophones.

The host connection is also useful in archiving speech data on the host. This data could later be retrieved for reanalysis on either the host or the PC, if the need arises.

Comparision with other systems

The following comparison between the present system and three other personal computer- or microprocessor-based systems does not show which system is good and which is not. All systems given here are good in terms of their own specific design objectives. We simply hope that this comparison highlights the features of each system, which make it more suitable than the others in a given application.

The comparison includes a system's technical aspects of the:

- processor and type of PC;
- speech segmentation method and graphics resolution needed;
- speech analysis method, analysis speed, length of speech utterance to be analyzed and synthesized at one time, and speech data compression rate;
- speech synthesis method;
- system utility for other purposes; and
- feasibility of connecting the system to a larger host computer.

These aspects are not absolute measures but are only indicators to the goodness of a system. To demonstrate this fact, we point out that analysis speed, for example, depends on—besides the speed of the processor—the particular analysis method chosen, the programming language used, and skill of the programmer.

The systems we have come across in the literature are:

1) a system developed by a group from the Philips

Company and centered on an HP9816S desktop computer,[10]

2) the Portable Speech Development Laboratory (PASS) from Texas Instruments,[11] and

3) the Interactive Laboratory System for the IBM PC and XT (ILS-PC1) from Signal Technology, Inc.[12]

Table 1 summarizes the technical aspects for each of the three systems and for our present system. As stated early in this article, the present system meets three objectives:

- a system to conduct experiments in synthesizing a language (Arabic is the main target) using a constructive-synthesis or a synthesis-by-analysis approach;
- a tool for conducting language-dependent studies, such as the study of prosodic features and the verification of the contextual variations in the synthesis units; and
- a tool for generating parameter codes, which later can be adapted to a specific target synthesizer.

The first aim is met by using the PC in a stand-alone mode. Fast editing and segmentation facilities, which can be used with low-resolution terminals, have been incorporated in the system. The analyses are carried out at reasonable speed by using only the power of the Intel 8088 (not a true 16-bit processor). As the system is experimental only, a software simulation of a synthesizer performs the synthesis. The study of the language-dependent aspects can be carried out only on the PC and at reasonable speed. The development of parameter codes to be used with a future target synthesizer requires the use of both the PC and a host at this stage.

The PASS and the Philips systems have been designed with different objectives than our present system. As far as I can see, both systems were designed

Table 1.
Technical aspects of present system
and other PC- or microcomputer-based systems.

System	Processor & PC used	Segmentation method & VDU resolution	*Analysis method *Analysis speed *Max segment length *Data reduction rate	Synthesis Method	System's utility for other purposes	Connnectivity to a larger system & host usage
The ILS-PC1	Intel 8088 Inside IBM-PC (not true 16-bit processor)	Speech waveform with a cursor VDU resolution not clear (possibly 640 x 200)	Spectral analysis other aspects not known	No synthesis	IBM PC on which system is hased can he utilized for other purposes	Possible, user provide his own communication hardware & software host used for speech data archiving
The PASS	Processor used is TM 9900 (16-bit) inside pass main System unit	VDU used in text mode for parameter edit No need for segmentation	*LPC analysis *Near real time *12 secs *96 kh/sec to 180 kh/sec	LPC synthesis by a hardware synthesizer	Special purpose system cannot he utilized for other purposes	Possible user provide his own software and hardware host used for speech data archiving
System based on the HP9816S	32-bit processor inside HP9816S	Speech waveform with a cursor VDU resolution 300 x 400	*Formant *45 sec/sec of speech *Other aspects not visihle to the author	Formant synthesis hy a hardware synthesizer	HP 9816S desk top computer can be utilized for other purposes	Not visihle to the author
The present system	Intel 8088 inside IBM PC	From short time energy and zero crossing rate VDU resolution 320 x 200	*LPC *300 secs/sec of speech *3.3 sec *120 kh/sec to 506 kh/sec	LPC synthesis by a software synthesizer	IBM PC & XT can he utilized for other purposes	Possihle user provides his own hardware & software host used for: 1. Further analysis 2. Speech data archiving

to act, mainly, as tools for developing parameter codes for specific target synthesizers using a synthesis-by-analysis method. The systems use either true 16-bit or 32-bit microprocessors. It appears, therefore, that in applications where the primary need is for fast development of coded speech to be used later in a product using a specific target synthesizer, the PASS system offers advantages over the others.

The Philips ILS-PC1 system is a general-purpose signal processing package, which has been adapted to run on the IBM PC and XT. It appears from the designer's choice of the PC and the wide range of signals for which the package can be used (from low-frequency underwater acoustic waveforms to very high frequency radar signals) that the aim was to satisfy the needs of a large class of scientists involved, in one way or another, in signal processing.

As far as I can tell, for speech analysis the ILS-PC runs at the normal PC pace (just like our present system), and it allows users to perform spectral analysis and encode speech in terms of formants. It appears also that the present implementation of the ILS-PC does not support speech synthesis, as was the case with the ILS for minicomputers.

IBM's Kuwait Scientific Center uses a speech segmentation, analysis, and synthesis system, which is centered on the IBM PC XT. This system can be used in applications in which users wish to experiment with language synthesis and language-dependent studies before committing to a target synthesizer.

The system can also be used to develop speech parameter codes that could eventually be adapted to a specific target synthesizer. In a stand-alone mode the system performs functions useful in speech synthesis research. These include the:

- input and editing of speech;
- creation of phonetic speech segments, words, phrases, or inventories of sentences and measurements of their durations;
- analysis and encoding of speech in linear predictive codes;
- study of prosodic language features translated into displays showing pitch and energy contours;
- verification of contextual variations of the synthesis units and measurements of their durations and changes in LPC spectral properties; and
- software simulation of a speech synthesizer as the discrete-time linear predictive model for speech production.

These functions are enhanced by having access to a mainframe computer supporting a general-purpose signal processing package such as the IAX. Such a connection offers additional signal processing capabilities, for example, coding speech by using formants and formant-based studies of a language. The connection can also be used for archiving speech data on a host.

The menu-driven system is fully user-interactive and modular. Modularity permits the system to be changed easily to accommodate signal processing algorithms other than those already implemented on it. As it stands now, the system uses only the Intel 8088 microprocessor, which is the heart of the IBM PC XT computer. Its speed in performing functions is good. However, system performance could be enhanced further by (1) adding the Intel 8087 coprocessor, (2) changing the existing software and hardware to an IBM PC AT with a true 16-bit processor, and (3) adding dedicated speech processing hardware such as boards with hardware signal processing capabilities. Besides increasing the speed of the existing software, any of these methods can also turn the system into a very powerful and speedy formant analyzer and synthesizer.

The system has been developed as part of a research project in speech synthesis. This project envisages synthesizing an unlimited Arabic vocabulary by using a constructive-synthesis approach (using short speech segments such as demisyllables or allophones). So the system must, at some stage, have the capability of turning Arabic orthographic input into speech. Provision has been made for this capability in the design; the modularity of the system also helps.

Currently, the system verifies the results of a phonetic study of the Arabic language, which is determining and validating the synthesis units. Very good results have been achieved with a syllabic approach to synthesis.

Finally, we compared this system with other systems of a similar nature. We showed its potential use as an experimental research system in language constructive synthesis and in the study of language phonetic properties.

References

1. R.W. Schafer and L.R. Rabiner, "System for Automatic Formant Analysis of Voiced Speech," *J. Acoustic Soc. America*, 1970, pp. 634-648.
2. M.R. Schroeder, "Vocoders: Analysis and Synthesis of Speech," *Proc. IEEE*, Vol. 54, Los Alamitos, Calif., 1966, pp. 720-734.
3. J. Makhoul, "Linear Prediction: A Tutorial Review," *Proc. IEEE*, Vol. 63, Los Alamitos, Calif., 1975, pp. 561-580.
4. J.D. Markel and A.H. Gray, Jr., *Linear Prediction of Speech,"* Springer-Verlag, New York, 1976.

5. E.M. Hofstetter, "An Introduction to the Mathematics of Linear Predictive Filtering as Applied to Speech Analysis and Synthesis," MIT Lincoln Labs tech. note, Cambridge, Mass., 1973.
6. C.H. Coker, "Speech Synthesis with Parametric Articulatory Model," *Proc. Speech Symp.,* Kyoto, Japan, 1968.
7. L.R. Rabiner, R.W. Schafer, and J.L. Flanagan, "Computer Synthesis of Speech by Concatenation of Formant-coded Words," *Bell Syst. Tech. J.,* Short Hills, New Jersey, 1971, pp. 1541-1558.
8. D.H. Klatt, "Software for a Cascade/Parallel Formant Synthesizer," *J. Acoustical Soc. America,* 1980, pp. 971-995.
9. The ILS Software Package, Signal Technology Inc., 5951 Encina Rd., Goleta, CA 93117.
10. J.D. Ouden and M.T. Have, "Stand-alone Speech Development System Using a Personal Computer," *Proc. Speech Technology,* Media Dimension Inc., New York, 1985, pp. 288-291.
11. J. Helms and S. Peterson, "Portable Speech Development System Creates Linear Codes," *Electronics,* Sept. 1982, pp. 151-156.
12. *The ILS-PC Overview,* Signal Technology Inc., Goleta, Calif.
13. D. Bertrand, "A Speech Input and Processing Board for a Personal Computer, *Proc. ICASSP,* San Diego, Calif., 1984.
14. C. Hubert et al., "Speech Processing on a Personal Computer to Help Deaf Children," *IFIPS Conf.,* R.E.A. Mason, ed., Elsevier Science Publisher B.V. (North-Holland), 1983.
15. *PC-Mate Labmaster Users Guide,* Tecmar Inc., PC Product Division, 6225 Cochran Rd., Solon (Cleveland), OH 44139.
16. L.R. Rabiner and R.W. Schafer, *Digital Processing of Speech Signals,* Prentice-Hall, New York, ISBN 0-13-213603-1, Chap. 4 and 6, 1978.
17. S.H. Al-Ani, *Arabic Phonology,* Mouton, The Hague, 1970.
18. P. Jackson, "Experience with the IAX Image Processing System," Digital Signal Processing, V. Capellini and A.G. Constantindes, eds., Elsevier Science Publishers B.V. (North-Holland), 1984, pp. 255-261.
19. W. Koenig, H.K. Dunn, and L.Y. Lacy, "The Sound Spectrograph," *J. Acoustical Soc. America,* July 1946, pp. 19-48.
20. A.V. Oppenheim and R.W. Schafer, "Homomorphic Analysis of Speech," *IEEE Trans. Audio and Electroacoustics,* June 1968, pp. 221-226.
21. A.M. Noll, "Cepstrum Pitch Determination," *J. Acous. Soc. America,* Feb. 1967, pp. 293-309.
22. J. Bristow, *Electronic Speech Synthesis,* Chap. 6 and 9, ISBN 0-246-11897-0, Granada, London, New York, 1984.
23. P. Cummiskey, N.S. Jayant, and J.L. Flanagan, "Adaptive Quantization in Differential PCM Coding of Speech," *Bell Syst. Tech. J.,* 1973, pp. 1105-1118.
24. G. Fant, *Theory of Speech Production,* Mouton, The Hague, 1970.
25. J. Durbin, "Efficient Estimation of Parameters in Moving Average Models," *Biometrica,* Vol. 46, Parts 1 and 2, 1959, pp. 306-316.
26. M.M. Sohndhi, "New Methods for Pitch Extraction," *IEEE Trans. Audio and Electroacoustics,* 1968, pp. 262-266.

Yousif A. El-Imam joined IBM to work on computer-based speech analysis and synthesis. He has recently completed the implementation of a PC-based, unlimited-vocabulary, Arabic text-to-speech system. Previously, he worked for the World Health Organization in Geneva planning a local area network, for Kuwait University as a lecturer in digital electronics and microcomputer systems, and for L.M. Ericsson of Sweden as a telecommunication engineer on the AXE-10 computer-controlled telephone exchanges. His research interests are in speech synthesis and recognition, artificial intelligence, and computer communications and networks.

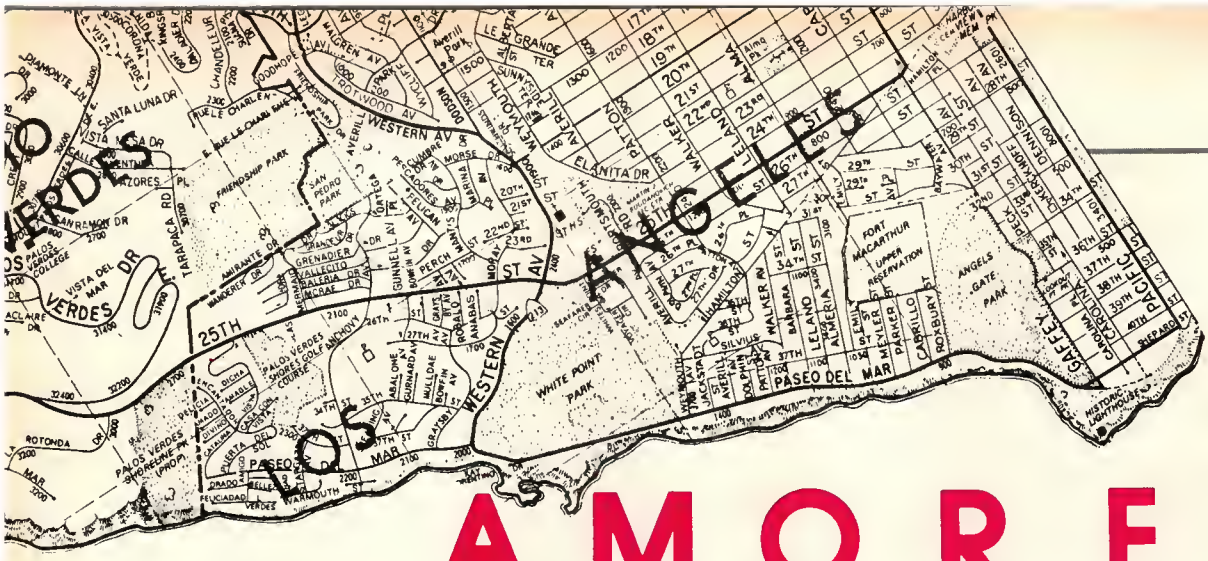
El-Imam received a BSc in electronics and a PhD in computers and controls from Dundee University, Scotland, in 1974 and 1978.

Questions concerning this article can be directed to Yousif A. El-Imam, IBM Kuwait Scientific Center, PO Box 4175, Safat, Kuwait.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 153 Medium 154 Low 155



AMORE

ADDRESS MAPPING WITH OVERLAPPED ROTATING ENTRIES

G.J. Dekker and A.J. van de Goor
Delft University of Technology, The Netherlands

A memory management unit can be designed specifically for use with the Unix operating system and can take the place of a commercially available MMU chip such as the Motorola MC68451 or the National Semiconductor NS16082. A team of students and staff members of the computer architecture group of the Laboratory for Switching Technique and Computer Architecture, of the Delft University of Technology in the Netherlands, has developed a general-purpose, high-performance, single-board system with such an MMU. At the time the MMU was designed, it had already been decided that the Unix operating system was going to be used and that the system would consist of a MC68010 CPU, 1M bytes of memory, and a fast Winchester disk.

A memory management unit that supports demand paging is implemented with standard logic and fast-access RAM chips, resulting in much faster address translation than that provided by the standard Motorola MC68451 MMU.

Memory management concepts. Let us begin with a functional description of a memory management unit. However, since our single-board computer is intended to be used with the Unix operating system, we will examine memory management in this context instead of giving a more general presentation.

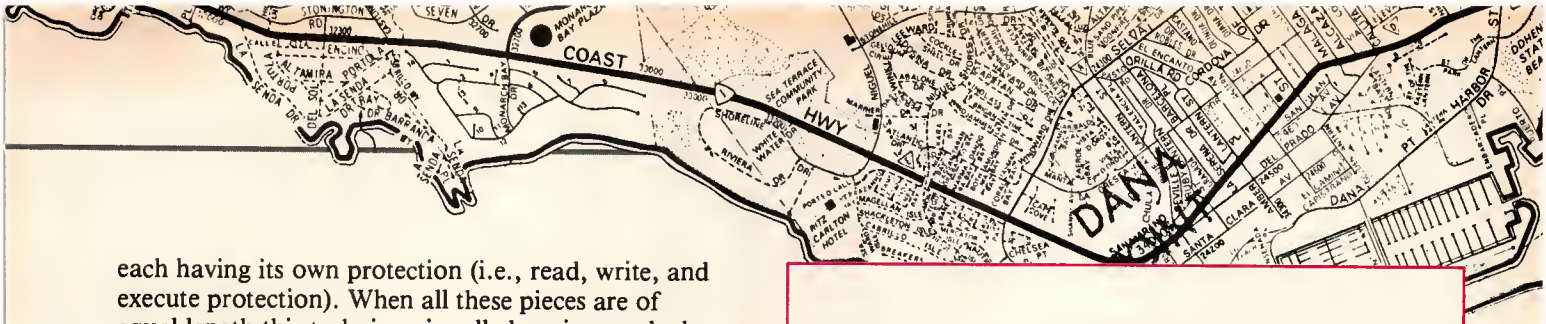
One of the most important components of a computer system (apart from the CPU) is the MMU.[1] This device maps addresses generated by the CPU, which are called logical addresses or sometimes virtual addresses (e.g., in the VAX), into addresses for the main memory, which are called physical addresses (Figure 1). The minimum tasks Unix requires this translation to perform can be described by the terms *relocation* and *protection*.

Relocation usually implies that sets of contiguous logical addresses are mapped to sets of contiguous physical addresses of an equal length but at a different location. Protection means that the MMU is capable of giving access protection to sets of logical addresses (that is, can specify read, write, and execute access).

Besides these two minimum requirements, two other capabilities are specified in more sophisticated versions of Unix—paging or segmentation and virtual memory.

It would alleviate the task of physical memory allocation strategies if it were not necessary for sets of logical addresses having the same relocation and/or protection to be mapped on single pieces of physical memory of equal length. Such mapping can be avoided if one subdivides a set of logical addresses into a number of partitions, each able to be relocated to noncontiguous pieces of physical memory and

0272-1732/87/0600-0022$01.00 © 1987 IEEE



each having its own protection (i.e., read, write, and execute protection). When all these pieces are of equal length this technique is called paging, and when they can be of varying length it is usually called segmentation.

Because of localities in address references, it is not necessary to have all pages or segments loaded in physical memory during the execution of a program. The operating system can load only those parts that are actually needed and store the remaining parts on disk. However, if the program generates a logical address located in a section that is not loaded in main memory, the MMU has to signal this fact to the processor, which postpones the partially executed instruction and takes the action required to load the appropriate section. After this, the program can be resumed. This scheme is called virtual memory.

Existing MMUs. Let us briefly examine three existing MMUs—the Motorola MC68451, the National Semiconductor NS16082, and the Western Electric Bellmac-32—and present the reasons why we did not choose any of them.

The MC68451. Since this MMU did not seem well suited to our system, we will not elaborate on its details.[2] However, it had the following advantages:

- It is a member of the M68000 family, which makes it easy to interface to the MC68000.
- It is a single-chip VLSI design, which makes it reliable and easy to implement.
- The version of Unix we were using had already been adapted to it.

But the MC68451 also had several disadvantages:

- It allows only 32 variable-length segments to be used per MMU.
- It forces each segment to have a length equal to a power of two, complicating memory management strategies (binary buddy algorithms, etc.).
- Its fastest version (8 MHz) still has a 217-ns address delay time, which introduces two extra wait cycles in a memory access.
- It is not very well suited for some kinds of virtual memory due to its limited number of segment registers (although some form of demand segmentation can be implemented).

It was these disadvantages that made us decide not to use the chip.

The NS16082. This MMU supports 32-bit, demand-paged virtual memory architectures.[3] It uses a combined segmentation/paging scheme to support the virtual memory architecture of the NS16032. Because this chip offers a mechanism which seemed ideal for our system, we will briefly discuss the memory architecture it provides.

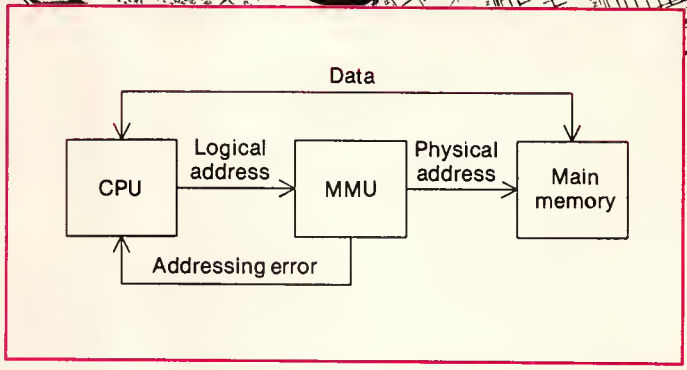


Figure 1. Logical to physical address mapping through an MMU.

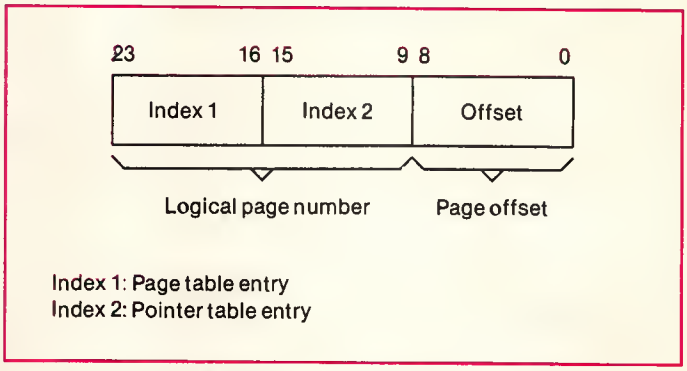


Figure 2. The logical address in the National NS16082.

The 16032 CPU has a logical address space of 16M bytes and a physical address space of 16M bytes, and each is partitioned into 512-byte pages. To minimize the mapping table size, designers used a two-level approach. A logical address consists of two fields: an offset within the 512-byte page and a logical page number (Figure 2). This logical page number is partitioned into two subfields: an eight-bit page table entry address (Index 1) and a seven-bit pointer table entry address (Index 2). The eight-bit page table entry address specifies an index address for the first-level page table, which has 256 entries of 32 bits each. The MMU has two page table base registers, PTB1 and PTB2, one for the user and the other for the supervisor. Index 1 indexes into the page table, which has its base in PTB1 or PTB2. Each of the 256 page table entries contains a 15-bit physical page frame number that selects one of 256 pointer tables. Each pointer table has 128 pointer table entries. Index 2 indexes into the pointer table. Each pointer table entry is 32 bits wide, and each generates a 15-bit physical page frame number. The least significant nine bits of virtual address are appended with this 15-bit physical

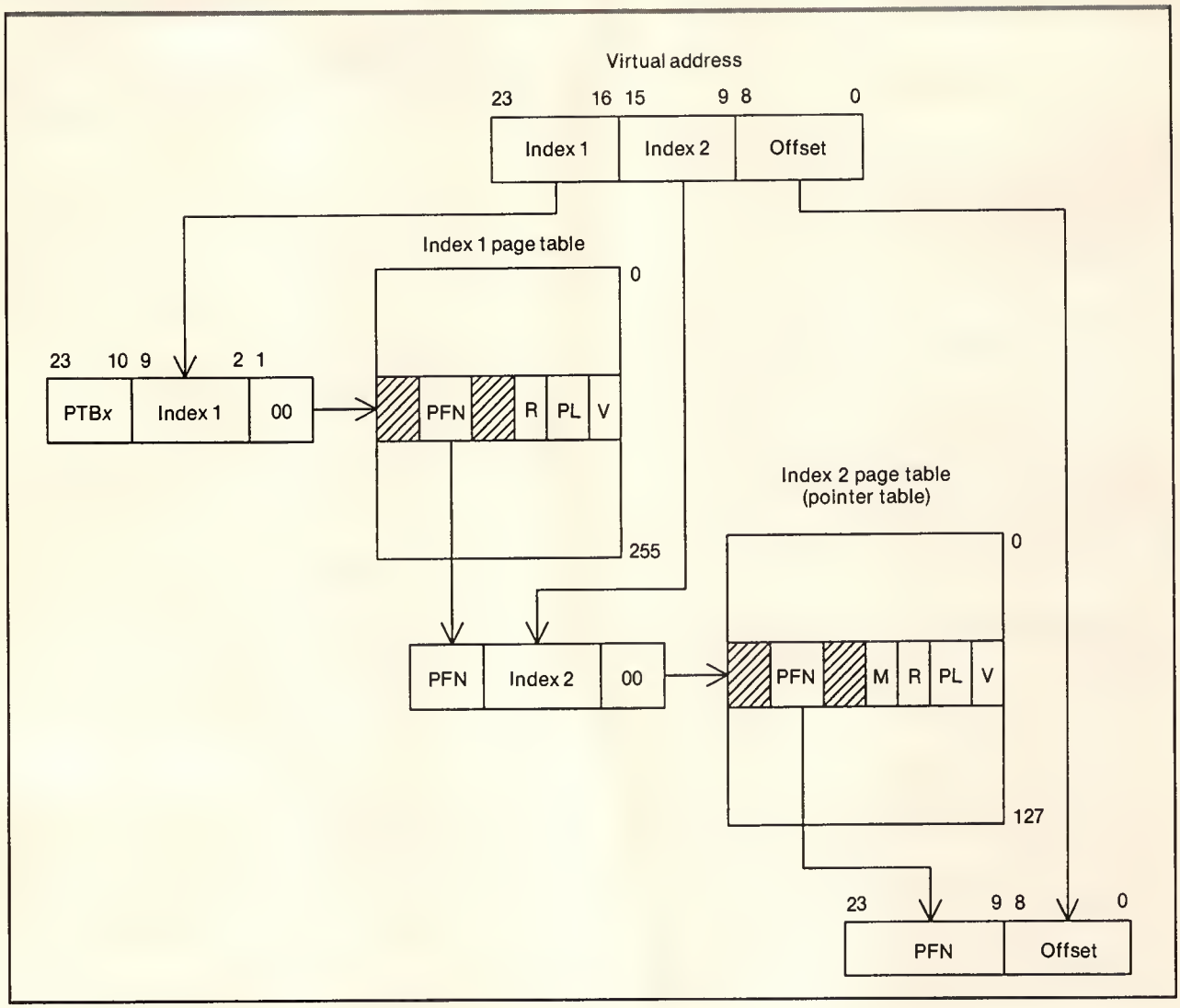


Figure 3. Logical to physical address translation in the NS16082.

page frame number to generate the 24-bit physical address. The address translation process is shown in Figure 3. The MMU has 32 entries in its translation buffer (associative cache). If the associative compare results in an address hit, the translated address is generated quickly. However, if there is no match, the MMU refers to the page table and pointer table in memory and tries to update the buffer.

Although the paging scheme implemented in the NS16082 seemed ideal for an operating system supporting paging, this MMU had drawbacks that made us reject it:

- It was rather expensive and difficult to obtain in fast versions at the time we were creating our design.
- It is difficult to interface to an MC68000, because its bus architecture is different from that of the M68000 family.
- Using this MMU would have introduced a considerable extra address delay time, due to the interface logic that would have been needed.

The Bellmac-32. The Bellmac-32 MMU[4] and the NS16082 are comparable in functionality, except that the NS16082 supports paging only. Both have hardware to perform miss processing in an on-chip descriptor cache. A comparison of the Bellmac-32 and the MC68451 shows that the latter supports segmentation only (it treats paging as a special case of segmentation) but that it allows multiple MMUs to be connected to a single CPU.

The Bellmac-32 MMU has the additional capability of supporting paged and unpaged segments. However, because this chip was not available when we started our project and because the rotate mechanism we describe here seemed very promising, we did not further consider the Bellmac-32.

Because existing single-chip VLSI MMUs either didn't meet our requirements or weren't yet available in working silicon, we decided to build our own MMU, realizing that this would not only enable us to meet our requirements but also give us experience in an important area of computer architecture.

Memory management on the VAX-11/750

We had a VAX-11/750 with Berkeley Unix 4.1 in our laboratory, and we felt it could be useful to study the logical memory features of that architecture. We felt that if we could in some way simulate the VAX's MMU mechanism, we could adapt Berkeley Unix to our single-board computer.

Hardware. Here we will summarize the part of the VAX architecture that relates to memory management, as it is described in the VAX handbooks.[5,6] Figures 4, 5, and 6 illustrate some important features of the VAX memory architecture.

The logical and physical address spaces are divided into 512-byte pages. Of the logical address space, one half (that with the most significant bit set) is referred to as *system space*. System space contains the operating system software and system-wide data, which is shared by all processes. The other half of the logical address space is defined for each process; it is therefore referred to as *process space*. Process space is further divided (on the next most significant address bit) into P0 space, in which program images and most of their data reside, and P1 space, in which the system allocates space for stacks and process-specific data. Because the P1 space is used for stacks, it is allocated from high addresses downward. Each process has its own P0 and P1 spaces, independent of others in the system. Figure 4 shows the address spaces of several processes. Each process space is independent of the others, while the system space is shared by all. Figure 5 shows the logical and physical address format, in which the size of the physical address is taken to be 32 bits long. (In fact, the width of this address is less.) The high-order two bits of a logical address immediately identify the space to which the logical address refers. Whether the address is physical or logical, the byte within the page is the same.

The processor has three pairs of page mapping registers for each of the three spaces actively used. These mapping registers are loaded by the operating system, along with the base address and length of the page tables. There is one active page table for each of the three spaces. A page table is a logically contiguous array of page table entries. Each page table entry represents the physical mapping for one logical page. To translate a logical address into a physical address, the processor uses the logical page number as an index into the page table from the given page table base address. Figure 6 shows the format of a page table entry. In concept, the process of obtaining a page table entry occurs on every memory reference. In practice, however, the processor maintains a translation buffer, which is a special-purpose cache of recently used page table entries. When one of the page tables is updated, this translation buffer must be invalidated by a special-purpose instruction.

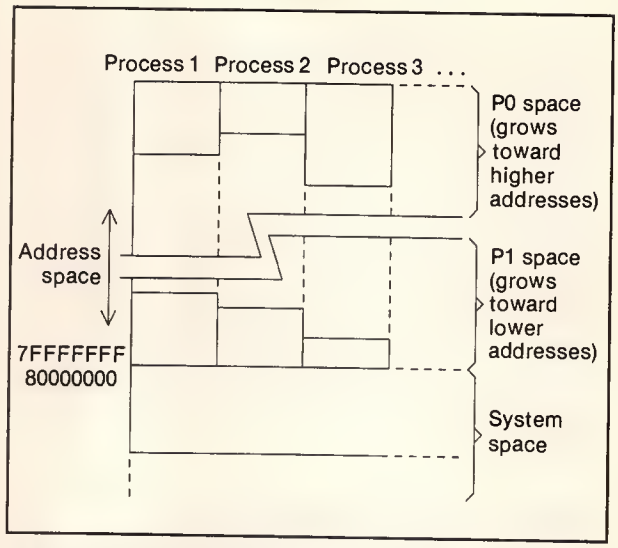


Figure 4. The address spaces of several processes in the VAX-11.

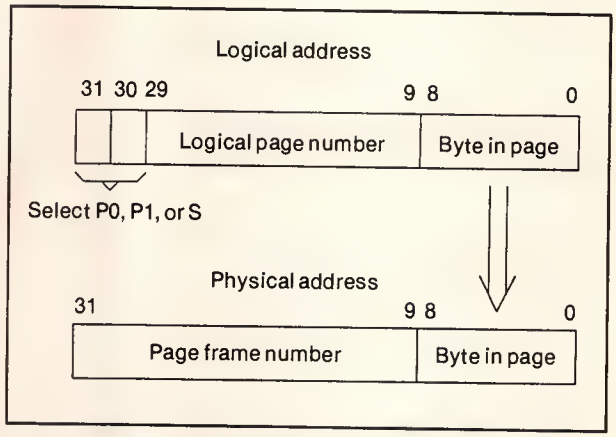


Figure 5. Logical and physical address formats of the VAX-11.

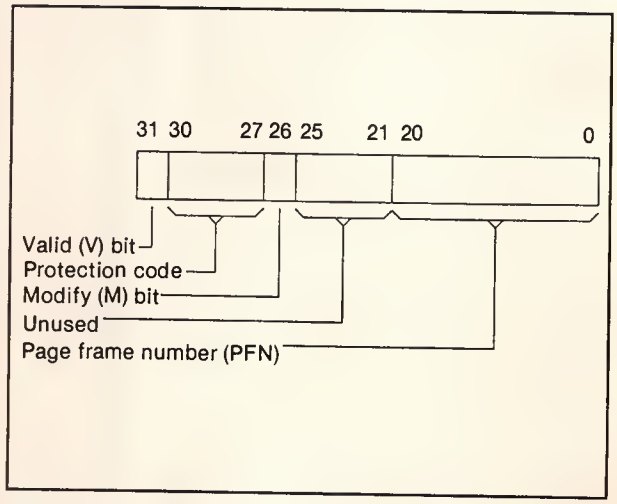


Figure 6. Page table entry format of the VAX-11.

Software. As mentioned before, the operating system used on our VAX-11/750 is Berkeley Unix 4.1, a descendant of the standard PDP-11 kernel implementation. Readers not familiar with this implementation can become familiar with it by studying Lions' commentary,[7] the Unix kernel models developed by Peppinck,[8] and the article on Unix implementation by Babaoglu and Joy.[9] With a thorough understanding of the nonpaging Unix kernel, one can go on to the excellent report on paging in Berkeley Unix by van Someren.[10] The requirements imposed on the hardware by Berkeley Unix will be discussed later in this article.

MC68010 paging support

Because we chose the M68000 family architecture for our single-board computer and thus had to develop our MMU to fit that architecture, we should examine those aspects of the MC68010 that relate to memory management.

The MC68010 is a VLSI, single-chip, 16-bit microprocessing unit with seventeen 32-bit registers.[11] It is fully object-code-compatible with the earlier members of the MC68000 family and adds virtual memory and virtual machine support, and it has improved instruction timing.

The processor operates in one of two states of privilege: the supervisor state or the user state. The privilege state determines which operations are legal, and it determines the choice between the supervisor stack pointer and the user stack pointer in stack references. It may be used by an external memory management device to control and translate accesses.

A bus error exception occurs when external logic terminates a bus cycle with a bus error signal. Whatever the processor was doing, it immediately begins exception processing. When a bus error occurs, a long stack frame (29 words) is used to save the entire state of the processor. This makes it possible to continue a partially completed instruction after the exception handler of the operating system has taken care of the memory reference problem that caused the bus error.

A special status word in the stack frame along with the fault address are used by the bus error exception handler to determine the memory location and function code at the time the bus error occurred. The RTE (return from exception) instruction is used to reload the processor's internal state. The faulted bus cycle is then rerun and the suspended instruction resumed.

The MC68000 is, in contrast to the MC68010, not capable of instruction continuation (which is required for operations such as block moves because a restart is not possible), and thus is not well suited to virtual memory applications.

MMU requirements

We wanted to use the Berkeley Unix 4.1 operating system on our single-board computer and, if possible, adapt it to our own paging memory management architecture. Here, we will summarize the requirements that Bsd 4.1 imposes on memory management hardware and point out some other design objectives that have to be met.

VAX compatibility. Bsd 4.1 does not use all the memory management features of the VAX architecture, and it changes some features—and adds others of its own—in a software layer it places around the hardware. We will discuss the essential details.

Address spaces. The mode of the processor (supervisor or user mode) does not influence the mapping performed by the memory management of the VAX and is only used for access checking. This results in only one logical address space formed out of the P0, P1, and S spaces. Although the P0, P1, and S spaces in the conventional VAX architecture each can be as large as 1G bytes, Bsd 4.1 restricts the maximum size in the following way:

- The P0 space, which contains the user mode text segment and data segment, is restricted to 12M bytes (6M bytes for the text segment and 6M bytes for the data segment). However, we have found no Unix application programs that grow over our VAX's 2M-byte physical memory limit, although we could easily write a program exceeding this limit.
- The P1 space contains the user mode stack segment, which is restricted to 6M bytes, and the user structure and kernel mode stack segment, which together occupy 4K bytes. As with the P0 segment, the 6M-byte limit is way beyond the requirements of normal programs.
- The S space contains the kernel code, kernel data, and a number of different subspaces. It requires almost 400K bytes.

Closely connected to the VAX hardware are the page table structures. Our MMU has to recognize similar page structures (e.g., a P0, a P1, and an S page table). The entries in these tables contain at least the following fields:

- A valid bit (V), which indicates that the hardware is allowed to use the remaining fields of this format.
- A referenced bit (R), which actually is not supported by the VAX architecture but simulated by software at the expense of considerable overhead.[9]
- A modified bit (M).
- A protection field (PROT). Bsd 4.1 needs only a supervisor write access bit, a user write access bit, and a user read access bit.
- A page frame number (PFN).

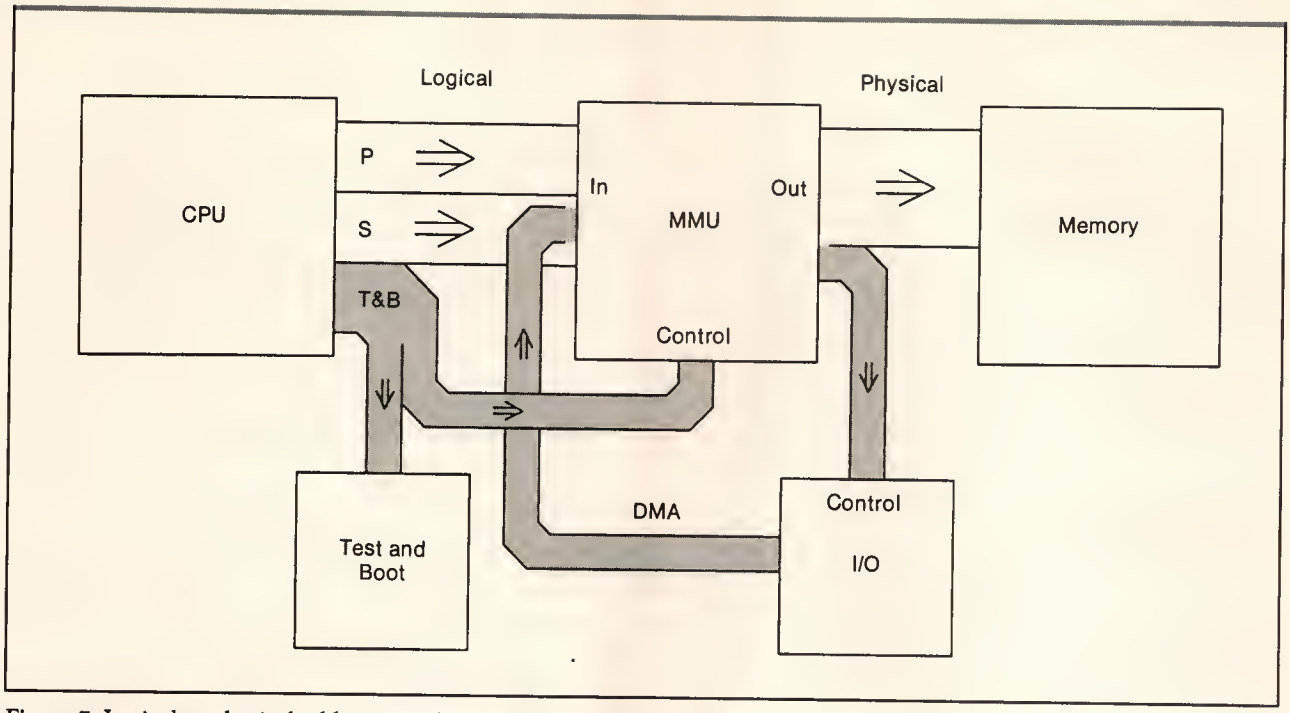


Figure 7. Logical to physical address translation in our approach.

Page size. Although the VAX hardware supports 512-byte pages, Bsd 4.1 normally treats multiple hardware pages as one unit, thereby enlarging the actual page size and raising performance.[9] On our VAX system, with its 2M bytes of installed physical memory, the actual page size (determinable at system compile time) was chosen to be 1K bytes. A page size of 2K bytes seemed the right choice for our single-board system, since programs and installed physical memory sizes tend to grow.

DMA support. The governing design rule for our project was to build a system with a high performance/price ratio. Implementation as a single-board system, to eliminate a backplane and bus interface logic, was a logical consequence. Another consequence of this rule was that the DMA controllers[12] used in our system are only capable of generating physical memory addresses in a 64K range. Because all system logic is concentrated on a single board, it is possible to give the MMU a double function: translating the addresses from the CPU and translating the addresses generated by the 64K DMA channels. A bus arbiter is required to merge the two address/data streams. The 64K address range coming from a DMA controller can be mapped by the MMU onto any address range within the physical memory space. Moreover, if the MMU is implemented with a paging mechanism that is also used for the mapping of DMA addresses, scatter/gather I/O can be performed because the I/O is done in the logical address space.

Test and boot space. Another design objective is testability, which means that the system must be able to test itself during the power-on process and possibly indicate a faulty unit on the system console. If the MMU is not functioning correctly, the processor cannot access memory and I/O (if these are mapped on physical space).

If one reserves a part of the logical address space for a so-called test and boot space, one can solve this problem. This test and boot space, which is addressed directly in logical space (without relocation or paging), contains test and boot software in read-only memory and the system console terminal interface. With this arrangement, the system can check the MMU function and all units that are accessed through the MMU (e.g., memory and other I/O devices) and report the results to the system console. Another advantage of this approach is that upon power-up of the system, the MMU mapping must be initialized. This action can be done in hardware, but it is cheaper to have the software perform this task. Upon power-up, the processor receives a RESET signal and fetches a restart vector containing an initial program counter and a supervisor stack pointer. These first four logical address references are forced to come from a special bootstrap ROM and point to a bootstrap program located in the test and boot space. This program can initialize the MMU to the desired mapping before any of the logical addresses that are mapped by the MMU are used. This requires accessibility to the MMU in the test and boot space. Since the test and boot space is not under control of the protection mechanism of the MMU, special hardware must ensure that an address reference falling in the test and boot space is legal only in the supervisor state. In the user state a bus error must abort the address cycle. Figure 7 shows this logical to physical address translation scheme.

MMU architecture

As we have seen above, the MMU will actually have to support three spaces (P0, P1, and S), whereas the part of the logical address space occupied by the test and boot space will need no mapping besides the rudimentary protection mechanism that allows access only when the processor is in the supervisor state.

Our MMU accommodates P0, P1, S, and test and boot spaces in the total logical address space of 16M bytes (Figure 8). The P0 space (for code and data) and the P1 space (for the stack and user structure) are conceived as a single P (private) mapping consisting of 2K pages, each 2K bytes long, for a total of 4M bytes. The S (shared) space also consists of 2K pages of 2K bytes each, totaling 4M bytes. A part of this space is reserved for DMA purposes, although this has no consequences for the MMU. Four megabytes are reserved for the test and boot space, which is more than sufficient.

Shared space (S). The shared space is mapped by means of a conventional page table mechanism, whereby the table itself is contained in fast special-purpose memory chips residing in the MMU. The 4M bytes of the S space require 2K entries, each describing a 2K-byte page in physical memory. Figure 9 shows a logical address in the S space and its translation to physical space. The fields in the page table entries are

- INV, the invalid field (bits 29-24), in which INV = = 0 indicates a valid entry,
- UW, the user write access field (bit 23),

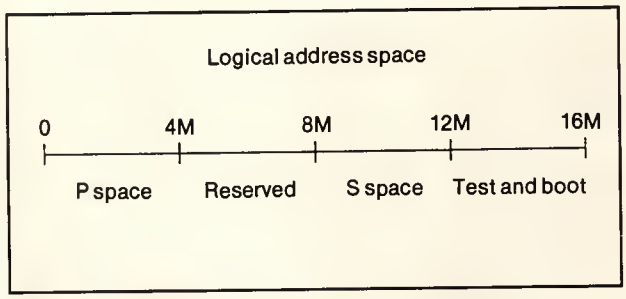


Figure 8. The logical address space in our MMU.

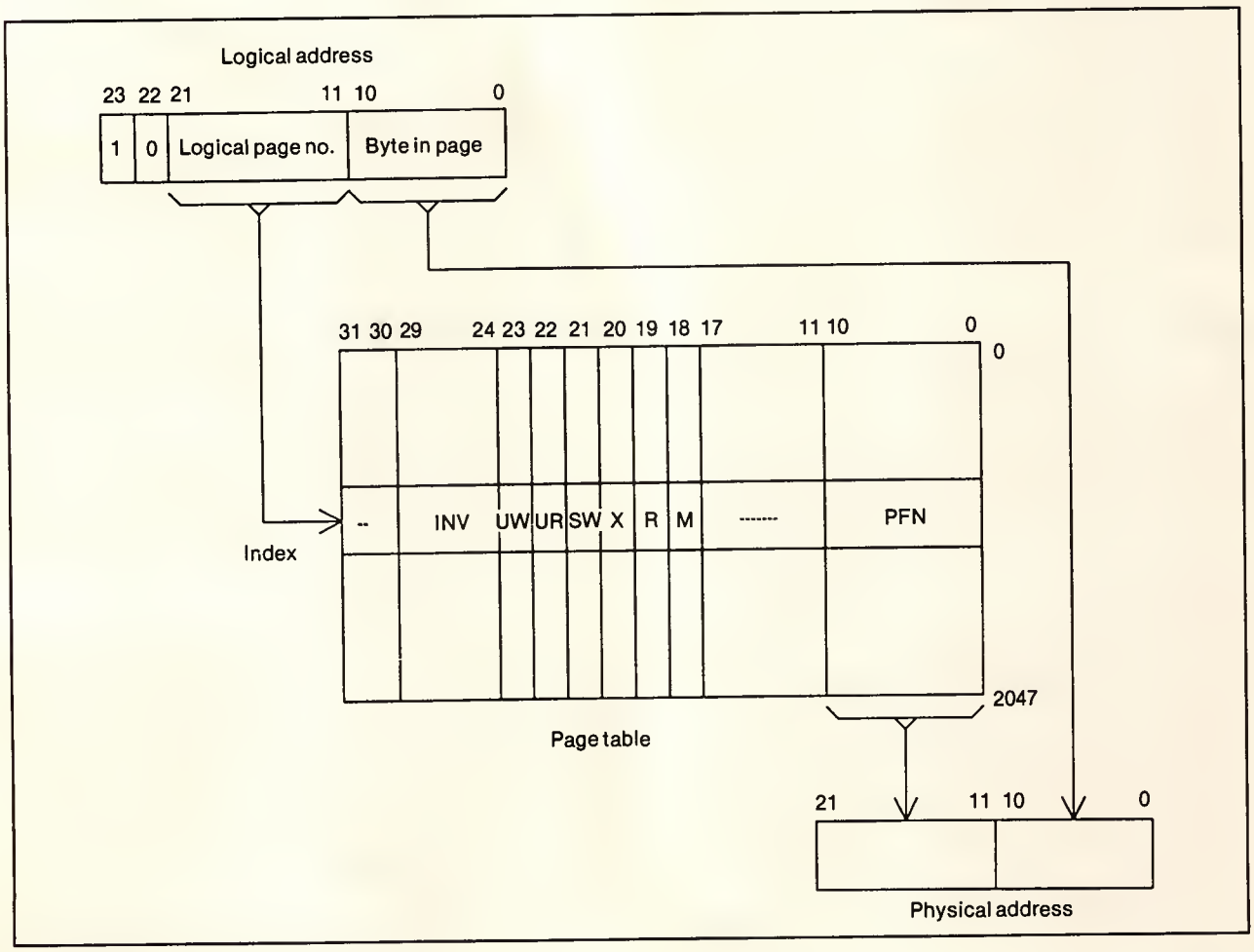


Figure 9. Mapping in the shared (S) space.

- UR, the user read access field (bit 22),
- SW, the supervisor write access field (bit 21),
- R, the referenced bit (bit 19),
- M, the modified bit (bit 18),
- PFN, the physical page frame number (bits 10-0), and
- X, the reserved bits (bits 31-30, 20, and 17-11).

The page table entries constituting the S space are accessible in the test and boot space as long words.

Private space (P). The P space occupies the logical address region from 0 to 4M – 1, and the next 4M addresses are reserved for future extension of the P space. The mapping used in the P space is process-dependent and must be altered on every process switch. In our MMU, the P0 space starts at logical address location 0 and grows upwards, and the P1 space starts at logical address location 4M – 1 and grows downwards (Figure 10).

Each process has its own page table for the P space in main memory. A memory management mechanism that could directly access these page tables located in physical memory would be very difficult to implement and would certainly require some kind of translation buffer. If we could provide a hardware page table like the one for the S segment for each 4M process space, we would only have to indicate to the MMU which table it has to use for the current process. Implementation of this mechanism would require a vast number of high-speed RAM chips, and it is not feasible with the present state of the art.

The rotate mechanism. Implementation of only one page table (which is feasible) would mean that many entries could be supplied with a new value on each process switch. Each entry that is invalid in both the current process and the new process could remain unchanged. However, all entries that are used by the new process, and those entries that are unused by the new process but were used by the current process, would have to be changed. In the worst case, this would mean that 2K entries would have to be changed, which would be quite time-consuming, especially if context switching were frequent.

In our MMU, we found a compromise between the last two solutions. One should realize that of the possible 2K entries describing 4M bytes, only a small amount will be used by most programs. Measurements at our laboratory have indicated that the typical Unix program is small (that is, that P0 is 32K bytes and P1 is 8K bytes). Many entries in most programs therefore will have to be initialized to invalid. Furthermore, it is possible to limit the maximum number of processes—say 64—that can be loaded into core (swapped in). If we have 64 programs with an average size of 64K bytes (i.e., requiring 32 entries), 2K hardware page table entries will suffice,

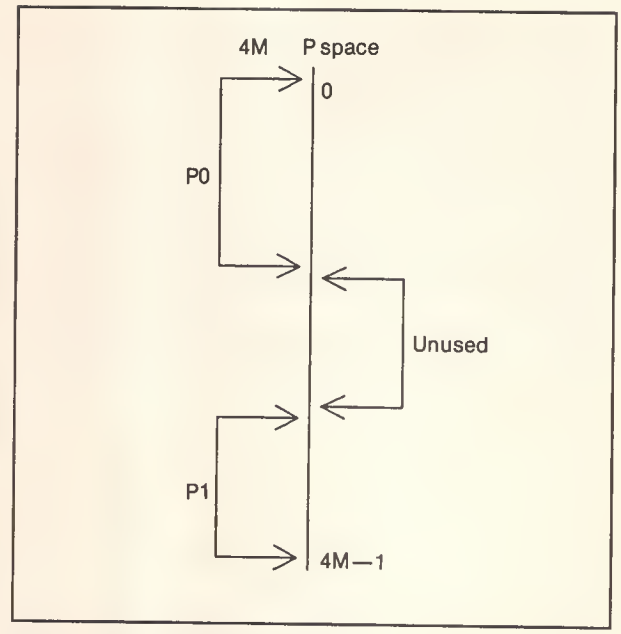


Figure 10. The private (P) space.

when entries are allocated as optimally as possible. A step in this direction is made in the way shown in Figure 11. The hardware page table is indexed with the logical page number plus an offset corresponding to the current process number multiplied by 32, so that indices higher than 2047 will be wrapped around (rotated).

The problem is that processes can need page table entries that are already being used by another process. This is solved by giving the hardware page table entries an additional process number field and by supplying the MMU with a current process number register (also required for the appropriate offset value) that is loaded by the software with a new value on each process switch. When the current process generates an address resulting in a page table entry with a process number that does not correspond to the current process number, an abort signal is generated to the processor. The bus error exception handler routine recognizes this situation and fills the faulting entry. The MMU, along with the software layer, functions as a cache for the actual page tables in main memory and is comparable to a direct mapping mechanism.[13] The resulting addressing mechanism is depicted in Figure 12. The page table entry contains the same fields as the one in the S space, except for the INV field, which is named PSN# (process number).

The mapping process proceeds as follows:

(A) The MMU combines the process number and the logical page number into an index in its hardware page table.

(B) If the page table entry contains a process number matching the current process number, the mapping process is continued at step D, else step C.

(C) The MMU generates a bus error, after which the processor enters the exception handler routine. This routine determines from the exception stack frame the faulting logical address and the intended action. It now fetches the software page table entry for the page containing the faulting address. If the intended action is illegal according to this entry, step E is taken; otherwise, it means that the MMU faulted because of a process number mismatch. The appropriate entry in the hardware table is now filled and the exception handler returns, causing the aborted memory cycle to be rerun and resulting in steps A, B, and D! (The modified and referenced information contained in the replaced hardware table entry must be saved.)

(D) The process number matches, which means that the information contained in this page table entry is valid for this process. (Totally unused entries have the reserved process number 0.) If the intended action (read/write in the supervisor or user state) is not allowed according to the access bits, step C is

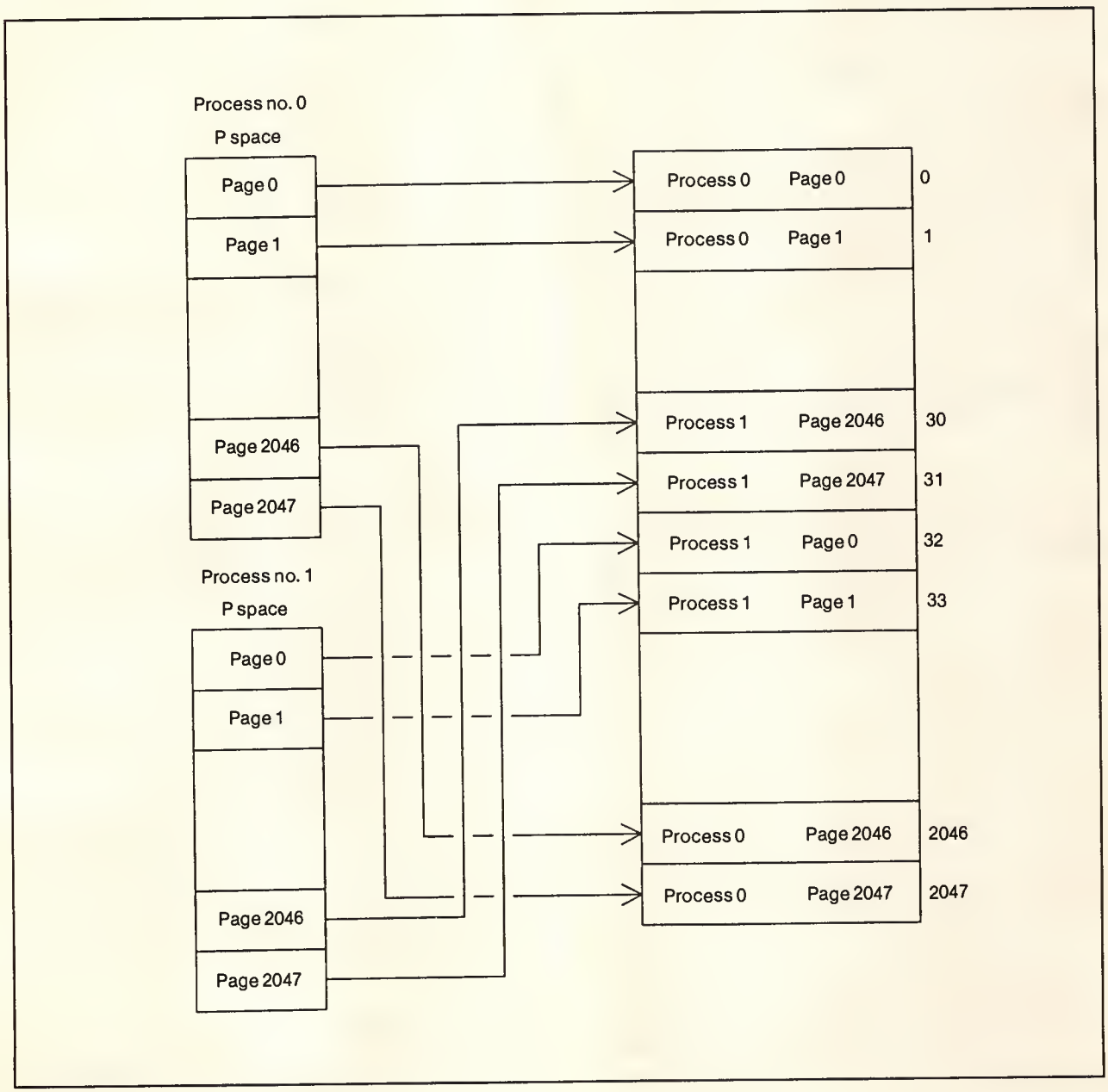


Figure 11. The P space with rotating entries.

taken. Otherwise, the modified (M) and referenced (R) bits are updated (*after* the process number match), and the physical page frame number (PFN) is combined with the "byte in page" address part of the logical address to form the *complete* physical address.

(E) A bus error is raised because of an access violation (indicated via the MMU status register) or because of a detection by the software that the desired page was not loaded in physical memory (i.e., that a normal page fault has occurred). When a bus error is raised, the appropriate actions are taken by the operating system. These actions are not relevant to the MMU at this point.

Performance considerations. The MMU hardware for the P space acts as a cache mechanism on the actual page tables in physical memory. From the fact that it contains only those entries for virtual pages that are actually loaded in physical memory, and from the fact that most processes will use only a small part of their maximum virtual address space, we can conclude that bus errors resulting from a cache mismatch are very rare. However, the action

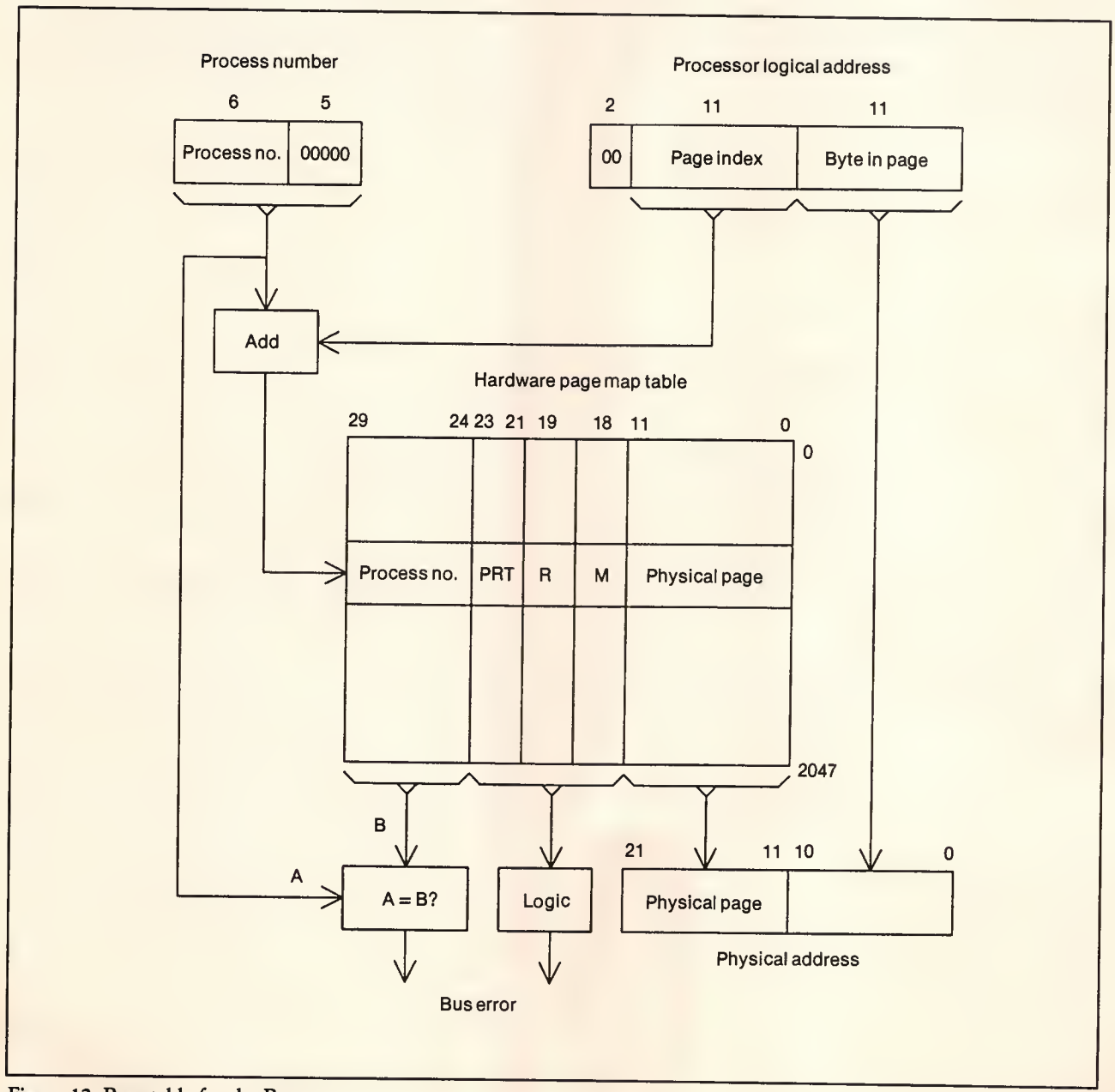


Figure 12. Page table for the P space.

needed on a cache miss is rather time-consuming (the equivalent of about 300 memory cycles or 150 ns). It consists of the following steps:

- bus error acceptance plus a vector load operation,
- pushing of the 29 words forming the long stack frame of the MC68010,
- the saving of the registers that are going to be used by the exception handler routine,
- the determination of the appropriate hardware page table entry according to the faulting address stored in the exception stack frame,
- the recognition of a cache mismatch fault among other fault conditions such as protection violations,
- the fetching of the length and location of the appropriate software page table and the selection of the right entry if this entry is valid,
- the restoring of the entry in the hardware page table,
- the reloading of the saved registers, and
- the resumption of the aborted instruction according to the exception stack frame (i.e., the popping of the 29 words).

The worst-case procedure is performed if, due to actions in the past, the MMU cache no longer contains entries for the just-restarted current process. If N virtual address pages are referenced (with each page able to be referenced many times) during the time slice of this process, a maximum of N cache misses occur, resulting in N times 300 memory cycles. If, after a cache mismatch, a page does not appear to be loaded in physical memory, a page-in must be performed. After this page-in, both the software and hardware page table must be updated, but the overhead required for this is negligible compared to the total page-in time.

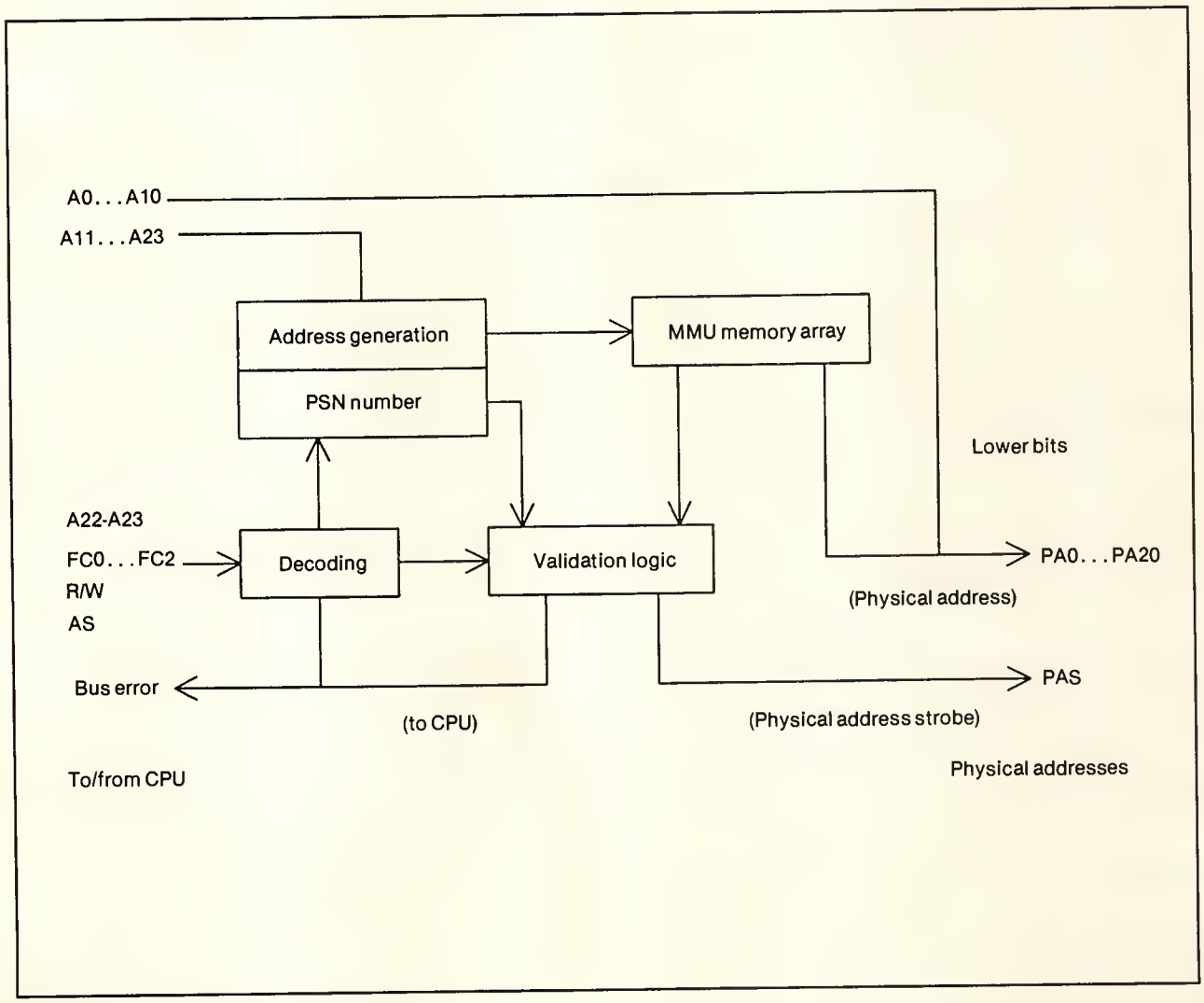


Figure 13. Block diagram of our MMU.

Test and boot space. The test and boot space occupies the region from 12M to 16M – 1. Logical addresses falling in this range are used to directly address physical memory (including memory-mapped I/O registers), which is bound in hardware to the test and boot space. The test and boot space mapping is predetermined in hardware and requires no initialization. Its "protection" mode is hardwired to allow accesses only when the processor is in the supervisor state. Objects addressable in the test and boot space cannot be accessed in the other spaces.

Implementation of the MMU

Here, we present a short description of the hardware implementation of the MMU; other details of the actual hardware design can be found in de Rijk.[14] Figure 13 illustrates the hardware implementation. The signals to the left are connected to the CPU, and the lines at the right lead to the internal physical address bus. We can see that the MMU is composed of four parts:

- A decoding section detects the appropriate space (P, S, or test and boot) of the current memory cycle and determines if this space is valid.
- Address generation logic selects the appropriate entry in the memory array. This section also contains the necessary program space number register (PSN#).
- A memory array holds the two page tables (one for the S space and one for the P space).
- Validation logic determines if the selected page table attributes are valid for the current address cycle. If they are not, a bus error is generated to the CPU.

The total address translation time is 90 ns.

Our MMU represents a useful alternative to existing single-chip implementations. Since the main goal of the single-board computer into which this MMU is integrated is to provide cheap processing power, the issues discussed here are more relevant than in the case of a single computer built to serve as a research vehicle. The RAM chips initially used in our MMU caused a total address delay of 120 ns, but faster, 45-ns versions have become available. By using these chips, we have reduced the total address delay time to 90 ns. The total number of chips we used to implement the MMU amounts to about 35, including the RAM chips. The total cost of the MMU's components is about $60.

Our MMU offers several advantages:

- It has an address delay of 90 ns compared to the address delay of 217 ns for the MC68451 and NS16082.
- It utilizes a paging technique for both the user and the supervisor space that results in efficient memory usage, especially when a virtual memory strategy is used.
- It makes a fast process switch possible.
- It can allow for permanently resident, shared libraries in the supervisor space, since the S space is visible to user programs and the page table entries in the S space contain a user read and write protection bit.
- It implements a reference bit, which is very useful when designing a paging strategy. (The VAX architecture lacks this bit.)

Moreover, the current single-board implementation of our system makes it possible to perform DMA through the MMU.

However, our solution also has a few disadvantages:

- No more than 64 processes can be swapped in, although this is not a severe restriction for a single-board system.
- The price of our MMU is no lower than that of existing single-chip MMUs, and in time the price of single-chip units will drop.
- Our design utilizes a large number of chips consuming a substantial amount of board space and power. The use of gate arrays could improve this, however.
- No Unix version exists that is already adapted to our MMU, although Berkeley Unix could be changed relatively easily to meet its requirements, provided enough physical memory is installed to hold the massive, permanently resident kernel code (138K bytes) and fast disks are used.

The current state of our project is that Unix System V has been modified to support the MMU in such a way that pages can be scattered throughout main memory during the time they have to be resident; that is, demand paging is not used but a scatter/gather approach is taken instead. This is a first step toward full demand paging, and it has been planned as a modification of a System V demand-paging release that is not available yet.

Several subjects have not been discussed here and could be investigated in future work:

- The hit rate of the MMU cache for the P space.
- Alternative page sizes.
- The allocation of process numbers to processes. If the total of 64 swapped-in processes is not reached, one could allocate process numbers with the largest gaps possible (e.g., two swapped-in processes could get process numbers 0 and 31).
- Hardware improvements (possibly using new chips). These could result in smaller delay times and fewer components.
- Hardware support for automatically loading a table entry upon a cache miss. This could be similar to that provided by the NS16082.

• Space extension. If the 4M-byte P space were extended to 8M bytes, it would be possible to extend the total number of swapped-in processes that could be allowed or increase the offset factor from 32 to 64 pages.

References

1. J. Peterson and A. Silberschatz, *Operating Systems*, Addison-Wesley, Reading, Mass., 1983.
2. MC68451 Memory Management Unit Data Sheet, Motorola, Inc., Austin, Tex., Sept. 1, 1979.
3. NS16082 Memory Management Unit Data Sheet, National Semiconductor, Santa Clara, Calif., Mar. 1982.
4. P.M. Lu et al., "Architecture of a VLSI MAP for BELLMAC-32 Microprocessor," *Digest of Papers—Compcon Spring 83*, Computer Society Press, Washington, D.C., pp. 213-217.
5. *VAX Architecture Handbook*, Digital Equipment Corp., Maynard, Mass., 1981.
6. *VAX Hardware Handbook*, Digital Equipment Corp., Maynard, Mass., 1980.
7. J. Lions, "A Commentary on the UNIX Operating System," and "UNIX Operating System Source Code, Level Six," Dept. of Computer Science, University of New South Wales, 1977.
8. W.R. Peppinck, "Modeling the UNIX Kernel," master's thesis, Laboratory of Switching Technique and Computer Architecture, Delft University of Technology, The Netherlands, 1984.
9. O. Babaoglu and W. Joy, "Converting a Swap-Based System To Do Paging in an Architecture Lacking Page-Referenced Bits," *Operating Systems Rev.*, Dec. 1981, pp. 78-86.
10. J. van Someren, "Paging in Berkeley Unix," internal report, Laboratory of Switching Technique and Computer Architecture, Delft University of Technology, The Netherlands, Feb. 1984.
11. *MC68010 Microprocessing Unit Data Sheet*, Motorola, Inc., Austin, Tex., 1981.
12. A.S. Tanenbaum, *Structured Computer Organization*, Prentice-Hall, Englewood Cliffs, N.J., 1984.
13. A.P. Pohm and D.P. Agrawal, *High-Speed Memory Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1983.
14. J. de Rijk, "Hardware Description of the SP Unix System," master's thesis, Laboratory of Switching Technique and Computer Architecture, Delft University of Technology, The Netherlands, 1984.

Further reading

"UNIX Time-sharing System," *Bell System Tech. J.*, July-Aug. 1978, Part 2.

H. Hellerman and T.F. Conroy, *Computer System Performance*, McGraw-Hill, New York, 1975.

G.J. Dekker is a member of the kernel support group of Associated Computer Experts BV, a software house in The Netherlands specializing in Unix. He is involved in porting parts of Unix that are dependent on particular machines to other machines having diverse MMUs, and he is helping develop a Unix local area network and a Unix windowing system. He holds an MSEE from the Delft University of Technology, The Netherlands.

A.J. van de Goor is a professor in computer sciences at the Delft University of Technology. Prior to this appointment, he served 11 years with Digital Equipment Corporation and IBM. His main interests are computer architecture, high-level-language machines, hardware testability, and multiprocessor systems. He received an MSEE degree from Delft University of Technology and MSEE and PhD degrees from Carnegie Mellon University.

Questions about this article can be directed to van de Goor at the Department of Electrical Engineering, Delft University of Technology (TH Delft), PO Box 5031, 2600 GA Delft, The Netherlands.

FEATURE

By implementing a capability-oriented addressing scheme, tagged storage, and a single-level-store approach to memory management, and by providing hardware support for multitasking, this architecture reduces the semantic gap.

THE ARCHITECTURE OF A CAPABILITY-BASED MICROPROCESSOR SYSTEM

Paolo Corsini, University of Pisa, Italy
Lanfranco Lopriore, Consiglio Nazionale delle Ricerche, Italy

Computer designers have often paid too little attention to reducing the distance between architectures and their operating environments.[1] This distance, the so-called semantic gap, is a problem that involves basic aspects of computer organization such as operating systems, programming language implementation, and programming environments. In particular,

- operating systems are poorly supported in their primary functions, such as memory management and protection, separation of privileges between tasks, interrupt servicing, and resource sharing,
- compilers receive little help in implementing basic concepts of modern high-level languages such as data abstraction and multitasking, and
- programmers obtain little assistance from the underlying machine in the test and debug phases of software development.

Here, we present the results of research aimed at the definition and implementation of a microprocessor-based advanced architecture whose main goal is the reduction of the semantic gap. This architecture is oriented toward high-level languages supporting modular decomposition of programs, user-defined data abstraction, and concurrency. Its salient features are a capability-oriented addressing scheme, an approach to memory management based on the concept of a single-level store, implementation of tagged storage by the tagging of memory segments, and significant hardware support for multitasking.

We present this architecture with particular reference to object types and memory management, and we evaluate it according to how well it reduces the semantic gap. We also show how it has been implemented as a research prototype in which the central processing unit has been built around an off-the-shelf microprocessor and in which an intelligent memory device autonomously supports the memory management functions.

0272-1732/87/0600-0035$01.00 © 1987 IEEE

The architecture

In our architecture, we have departed from the traditional von Neumann concept of a uniform, linear storage space. Instead, we follow an object-oriented, capability-based approach.[2,3] Essentially, tasks see the storage space as a single pool of objects. The architecture defines a set of mechanisms that make it possible to create and delete objects. When a new object is allocated, a unique identifier is assigned to that object. This identifier is never modified during the life of the object and, after the object has been deleted, is never used to name another object.

Objects are supported by a very large segmented virtual memory space. This space is partitioned into *areas*. Each implementation of the architecture (that is, each *machine*) is configured so it can allocate the segments of a specific area. Areas are provided mainly to support unique identifiers, even across the boundaries of a single machine. If any kind of information flow is to occur between two or more machines (through a network, or even just through a transportable storage medium such as a tape), those machines must be assigned to as many different areas.

Each object is contained entirely in a single segment, and the address of this segment in the virtual space is equal to the identifier of the object. Each segment is partitioned into three fields: a length field, a tag field, and an internal representation field (Figure 1). The contents of the length field specify the *size* of the internal representation field. The contents of the tag field specify the *type* of the object implemented in the segment. And the contents of the internal representation field represent the *value* of the object.

A task can access a given object only if it holds a *capability* for that object. A capability is an unforgeable, protected pointer that includes not only the object identifier, ID, but also an access right specification, AR. The architecture guarantees the integrity of capabilities, and the only modifications it allows are restrictions of access rights. However, it does permit capabilities to be freely copied so that access privileges can be transmitted.

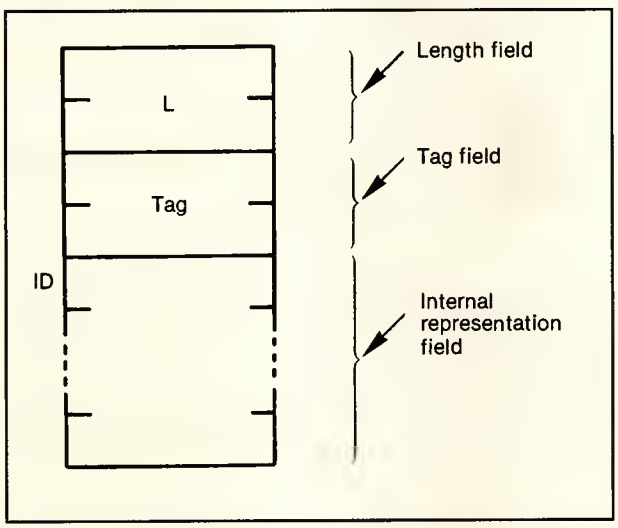


Figure 1. Virtual memory segment implementing an object whose identifier is ID. Both the length and the tag fields are at negative offsets, whereas the internal representation field extends to positive offsets, from offset 0 to offset L.

A set of virtual processors is implemented by the hardware of the CPU. This hardware performs the operations of one virtual processor at one time, and this processor is called the *running* virtual processor. Each virtual processor is provided with its own set of capability registers. A capability register can store a *long capability*. This is a quadruple {ID,L,Tag,AR} obtained by extending the capability {ID,AR} by means of the quantities L and Tag, which are contained in the segment implementing the object identified by ID. To reference an object in memory, a task attached to the running virtual processor must load a capability for that object into a capability register of the virtual processor.

Seven object types are supported by our architecture, and their operations are implemented by machine instructions. These types are the code space, the data space, the capability space, the task descriptor, the virtual processor image, the family factory, and the family root.

Code spaces, data spaces, and capability spaces. Code spaces store instructions in executable form. Data spaces allow all the usual arithmetical and logical operations to be performed on them. Capability spaces allow capabilities to be stored in memory. A capability space is a collection of capabilities. A capability in a capability space can be converted into a long capability and then loaded into a capability register to access the object it references. Conversely, a long capability in a capability register can be converted into the short format and then stored into a capability space.

Multitasking. Multitasking is supported by task descriptors and virtual processor images. The entire state of a single task (including the task stack) is stored in a task descriptor. The contents of a virtual processor image specify whether a task is attached to the virtual processor associated with that image. Attaching a task to a virtual processor means loading the state of that task from the task descriptor into the virtual processor. Detaching the task means copying the task state from the virtual processor into the task descriptor. In this way, different tasks can be attached to a given virtual processor at different times. The CPU can be caused to execute a given virtual processor, and this results in executing the task attached to that virtual processor at that time. As we will show later, the dualism of task descriptors and virtual processor images is aimed mainly at effective management of the set of virtual processors implemented by the CPU hardware.

Object types

Code spaces. A code space stores instructions in executable form.

Operations. A single action is defined on a code space—that is, the instructions contained therein. This action is made possible by the access right EXECUTE. The operations defined on code spaces relate to program control.

An example is the CALL operation. CALL transfers control to the instruction at a given offset of a given code space (Figure A). The old contents of the program counter are saved in a stack, the domain stack, which stores information about control transfers inside the same domain.

Data spaces. A data space is a collection of entries. Each entry contains a data value.

Operations. All the classical arithmetical and logical operations are defined on data spaces. An operation causing a reading or writing action on the contents of a given data space is made possible by the access

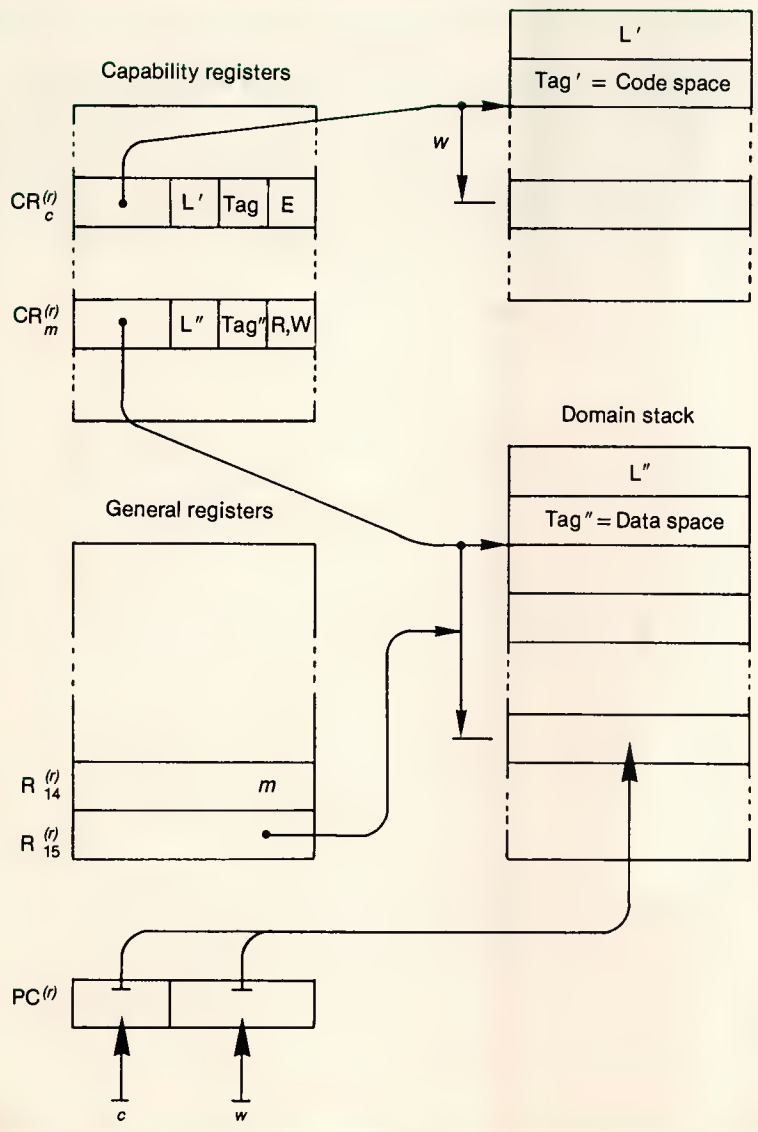


Figure A. Actions involved in the execution in the rth virtual processor VP_r of the call-to-subroutine instruction CALL $\langle\langle c \rangle\rangle$ w. This instruction transfers control at offset w of the code space addressed by $CR_c^{(r)}$ (i.e., the cth capability register in VP_r). The program counter $PC^{(r)}$ is partitioned into a segment number field and an offset field. The contents of the segment number field specify the capability register storing the instruction to be fetched next, at the offset specified by the contents of the offset field. The execution of this instruction loads the quantities c and w into the segment number field and the offset field, respectively. The old contents of $PC^{(r)}$ are saved in the data space addressed by the capability register, say $CR_m^{(r)}$, whose index m is specified by the contents of $R_{14}^{(r)}$. This data space implements the domain stack, and the top of the stack is pointed to by the contents of $R_{15}^{(r)}$.

right READ or WRITE for that data space.

Examples of operations for the data space type are LOAD and STORE. LOAD accesses the word at a given offset of a given data space and loads the contents of this word into a general register. This operation is made possible by the access right READ for the data space. STORE copies the contents of a given general register into the word at a given offset of a given data space. This operation is made possible by the access right WRITE for the data space.

Capability spaces. A capability space is a collection of entries. Each entry contains a capability.

Operations. The operations defined on capability spaces make it possible to move capabilities between capability registers and the main memory, and to transfer control from one subject to another.

LOAD_CAPABILITY converts the capability stored in a given entry in a capability space into a long capability. This long capability is then loaded into a capability register (Figure B). This operation is made possible by the access right TAKE for the capability space.

STORE_CAPABILITY converts the long capability contained in a given capability register into a capability. This capability is then stored in memory, into an entry of a given capability space. This operation is made possible by the access right GRANT for the capability space.

ACTIVATE_SUBJECT allows a subject S to activate another subject S′ of the same task on the same virtual processor. The state of S is saved into the task stack inside the descriptor of the task. This stack contains information about control switches across domain boundaries. The capability for the base of the domain of S′ is loaded into a capability register. Control is transferred at offset 0 of the code space referenced by the wth capability in the base of S′ (w is a parameter of the operation). Execution is made possible by the wth activate access right, namely ACTIVATE_w, for the base.

DEACTIVATE_SUBJECT transfers control back to the subject whose state is stored at the top of the task stack. The state of S is restored with quantities popped from the stack.

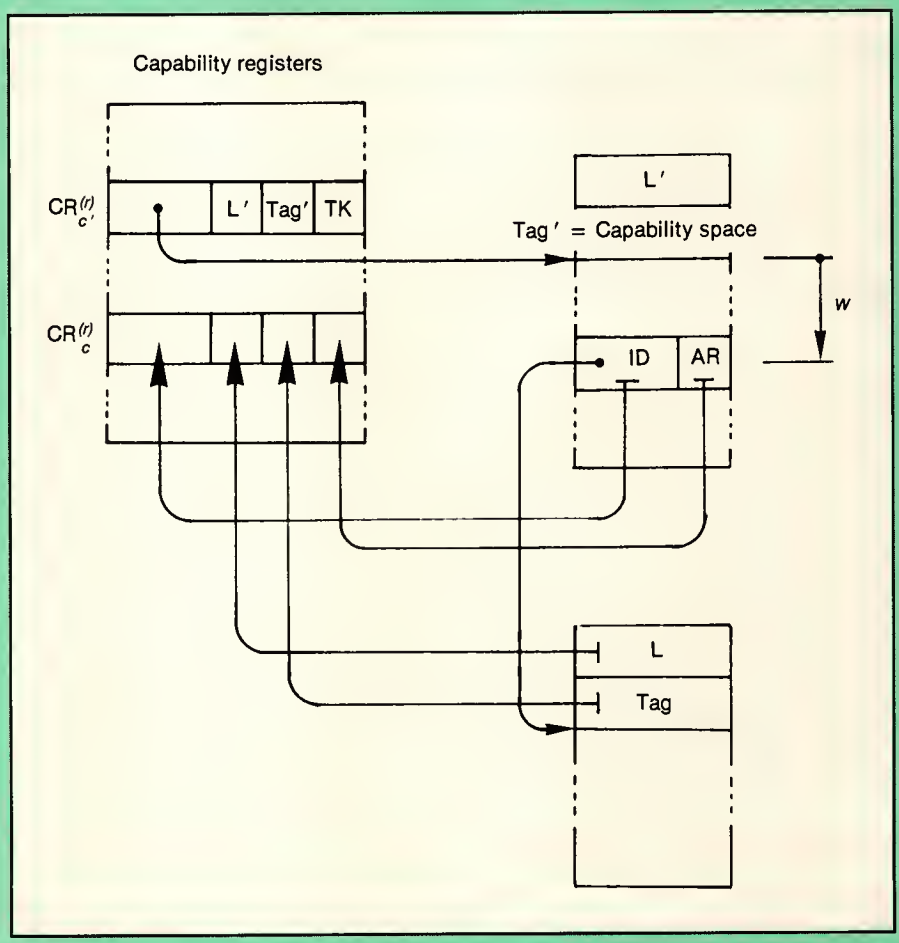


Figure B. Actions involved in the execution of the instruction LOAD_CAPABILITY #⟨⟨c⟩⟩, ⟨⟨c′⟩⟩ w. The capability stored in entry w of the capability space addressed by $CR^{(r)}_{c'}$ is extended by means of the contents of the L and Tag fields of the segment implementing the object referenced by that capability and then loaded into $CR^{(r)}_{c}$.

Virtual processor images. A virtual processor image contains the name of a virtual processor and a flag. This flag, if set, specifies that a task is attached to that virtual processor.

Operation. RUN__VP starts execution of the virtual processor associated with a given virtual processor image. This operation is made possible by the access right RUN for the image.

Task descriptors. A task descriptor stores the entire state of a single task (including the task stack).

Operations. ATTACH__TASK attaches a task to the virtual processor associated with a given virtual processor image. The state of the task is loaded from its descriptor into the virtual processor, and the flag inside the image is asserted. This operation is made possible by the access right ALLOCATE for the image and the access right ATTACH for the descriptor.

DETACH__TASK detaches the task attached to the virtual processor associated with a given virtual processor image. The state of the task is copied from the virtual processor into the descriptor of that task, and the flag inside the image is cleared. This operation is made possible by the access right DEALLOCATE for the image.

Family factory. A single object of the family factory type exists throughout the life of the system. At any given time, this object contains the name of the next family to be generated.

Operation. A single operation, GENERATE__FAMILY, is defined on the family factory. GENERATE__FAMILY allocates the family whose name is contained in the family factory. The contents of the factory are then incremented. Thus, the factory always contains the name of the next family to be generated. This operation, which is made possible by the access right GENERATE for the family factory, returns a capability for the root of the family allocated. This capability includes three access rights: ENABLE, USE, and INITIALIZE. These access rights are relevant to memory management.

Family roots. The root is the first object allocated in a family. At any given time, it contains the identifier of the next object to be created in that family.

Operations. After a root has been allocated by the GENERATE__FAMILY operation, it must be initialized by executing the INITIALIZE__ROOT operation. INITIALIZE__ROOT sets the root to contain the identifiers of the first object to be allocated inside the family after the root itself. This operation is made possible by a capability for the root with the access right INITIALIZE. The operation modifies the access right field of this capability to contain the whole set of the *create* access rights. These access rights permit execution of the create operations.

Five create operations make it possible to allocate and initialize new objects.

CREATE__CAPABILITY__SPACE allocates a capability space in the family of a given root and returns a capability for this space. This operation is made possible by the access right CREATE__CAPABILITY__SPACE for the root. The capability space is initialized to contain *null* capabilities (i.e., capabilities whose identifier fields are formed entirely of 1's). The operation may fail, however, and this occurs if the size of the capability space to be created is greater than the size of the residual free portion of the family.

CREATE__CODE__SPACE, CREATE__DATA__SPACE, CREATE__TASK__DESCRIPTOR, and CREATE__VP__IMAGE both allocate and initialize a code space, a data space, a task descriptor, and a virtual processor image. The actions involved in the execution of these operations are suggested by their names.

Four memory management operations allow the running task, say task T, to manage storage resources according to its own memory requirements.

ENABLE__FAMILY allows T to enable a given family. Execution of this operation fails if there is not enough free space in the bulk memory. This operation is made possible by the access right ENABLE for the root of the family.

USE__FAMILY allows T to state that it is a user of a given family. Execution causes an attempt to open the family. This attempt fails if there is not enough free space in the main memory and if there is no open family suitable for being swapped out (i.e., no family used by at most one process that is blocked). This operation is made possible by the access right USE for the root of the family.

RELEASE__FAMILY allows T to state that it no longer uses a family. The family is not swapped out immediately but will be swapped out eventually, when there is a lack of free space in the main memory. This operation is made possible by the access right USE for the root of the family.

REMOVE__FAMILY allows T to remove a family. Execution frees the memory areas reserved for storage of the family in both the bulk memory and, if the family is open, in the main memory. This operation is made possible by the access right REMOVE for the root of the family.

Protection domains. The capabilities inside a capability space may reference objects of any type and, in particular, other capability spaces. In this way, capability spaces can be organized into arbitrarily structured graphs. A protection domain is a graph shaped in such a way that all the objects it references can be reached by starting from a specific capability space (called the base of the domain), by means of the capabilities contained therein. When a task is attached to a given virtual processor, the capability registers of that virtual processor must store both the capability for the descriptor of that task and the capability for the base of a domain. These two capabilities together univocally identify the subject—that is, the pair {task, domain}—active on that virtual processor. A subject S can activate another subject S′ of the same task on the same virtual processor. The state of S is saved onto the task stack. Conversely, S′ may deactivate itself and transfer control back to the subject S, whose state is stored on the top of the task stack.

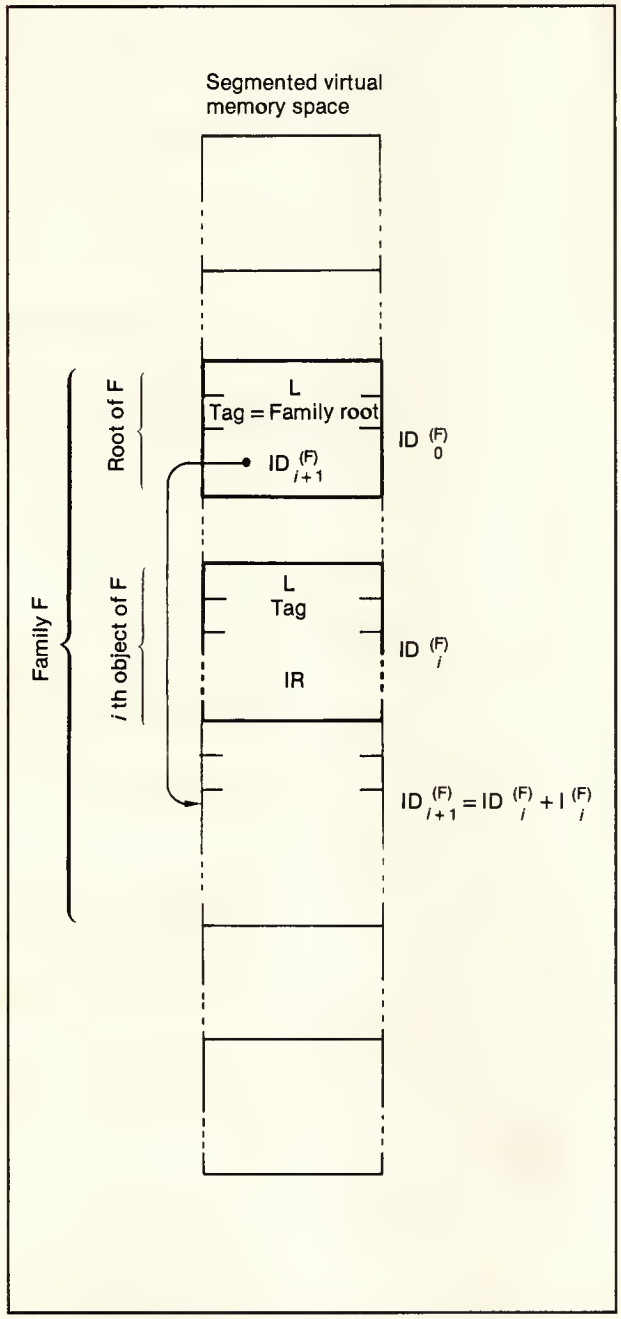


Figure 2. Organization of the segmented virtual memory space. The first object of a family F is the root of that family. At any given time, the root contains identifier $ID_i^{(F)}$ of the next object (say, the ith object) to be allocated in F. After allocation of this object, the root is updated to contain the identifier $ID_{i+1}^{(F)}$. This is given by the relation $ID_{i+1}^{(F)} = ID_i^{(F)} + l_i^{(F)}$, where $l_i^{(F)}$ is the dimension (in bytes) of the object identified by $ID_i^{(F)}$. However, allocation can actually be carried out only if no family overflow occurs.

Object creation. Memory management strategies are based on objects being grouped into *families*. Families are fixed-size units used for swapping between the bulk memory and the main memory as well as for garbage collection. A few families are generated by the bootstrap firmware; these are called *bootstrap families*, and the objects they contain relate to the resident portions of the system kernel. One such object is the *family factory*. At any given time, the family factory contains the name of the next family to be generated. Generation allocates the first object in the new family. This object, the *root* of the family, belongs to the *family root* object type; at any given time, it contains the identifier of the next object to be allocated in the family (Figure 2). A capability for the root makes it possible to allocate and initialize new objects in the family.

Memory management. A given family is *enabled* when a portion of the bulk memory is actually reserved for storage of that family. An enabled family is *open* when it also resides in the main memory; otherwise, it is *closed* (if it is stored only in the bulk memory) or it is *swapping in* or *swapping out* (if it is being copied from the bulk memory into the main memory or vice versa). An enabled family may be *removed* to free its portions of the bulk memory and, possibly, main memory; thereafter, all the objects in that family are lost (it will never be possible to reference them again). An object belonging to a given family can be accessed only if that family is open (always the case for a bootstrap family). A task may declare that it is a *user* of a given family. This means that once the family has been opened, it can be swapped out only if the task is its *sole* user and it is *blocked*.

Each task manages its own memory requirements by means of a set of operations—the memory management operations—that works on family roots. These operations allow the task to enable a family, become a user of a family, release (that is, cease using) a family, and remove a family.

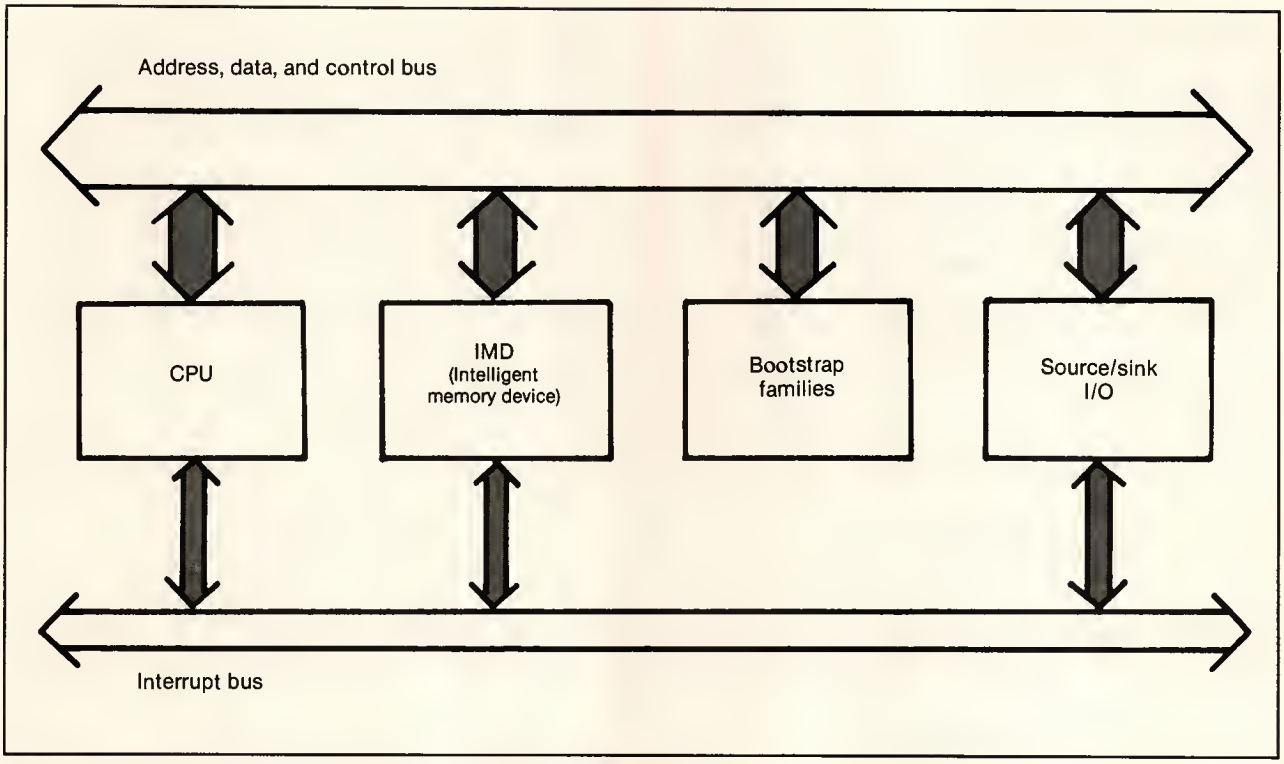


Figure 3. Hardware configuration of the prototype of the architecture.

Implementing the architecture

The hardware configuration of a prototype of our architecture is shown in Figure 3. A central processing unit accesses a pool of storage resources. These consist of an intelligent memory device (IMD) supporting memory virtualization, a set of read-only and read/write memory banks reserved for the storage of bootstrap families, and the memory-mapped interfaces of source/sink I/O devices (each interface being addressed in the same way as a data space in a bootstrap family).

The central processing unit. The CPU is based on a Zilog Z8001 microprocessor.[4] Ad hoc logics provide for capability processing and emulate the instructions implementing the operations of most object types. The instruction set consists of the following:

- All the standard instructions of the Z8001. Essentially, these implement the operations relating to code and data spaces.
- Special instructions, wholly emulated inside the CPU. These support the operations of all the other object types defined by the architecture, the only exceptions being the memory management operations.
- Memory management instructions. These implement the memory management operations. They are fetched by the CPU but are actually executed inside the IMD.

At any given time, the instruction to be fetched next is addressed by the contents of the program counter of the virtual processor running at that time. The program counter consists of a segment number field and an offset field. The contents of the segment number field specify a capability register in the set of capability registers associated with the virtual processor. This capability register references the code space being executed, and the offset of the instruction inside this code space is specified by the contents of the offset field. A similar mechanism is used to access an operand inside a data space. An address, as included in the instruction formats of the Z8001, consists of a segment number field and an offset field. The contents of the segment number field specify the capability register referencing the data space involved in the access.

Besides the program counter and the capability registers, the architectural interface of each virtual processor includes all the registers defined by the architecture of the Z8001. (The only exception is that the system/normal bit is not supported.) Moreover, a priority register specifies the priority of the task attached to the virtual processor. (As will be shown shortly, task priorities are relevant to interrupt handling.) Virtual processors are supported by a scratchpad read/write memory forming an array of capability

registers (Figure 4). Moreover, a read/write memory implements a save area for the contents of the general-purpose registers, the program counter, and the flag and control registers of all nonrunning virtual processors. Finally, an ad hoc read-only memory stores the firmware emulating the special instructions. When this firmware is being executed, the microprocessor runs in its supervisor state; therefore, the supervisor state is considered a microprogram state.

Interrupt sources are partitioned into *priority classes*. An interrupt request coming from a source belonging to a given class, say class C, is accepted only if the priority of the task attached to the running virtual processor is less than or equal to C. When accepted, the request is converted into a call to a specific task associated with class C, the *interrupt task*, and permanently attached to the Cth virtual processor. Let us examine this in greater detail. The name of the virtual processor that has been interrupted is pushed onto a firmware-handled interrupt stack inside the CPU. (This stack makes it possible to nest interrupts.) Then, the name of the interrupting source is stored into a general register; in this way, this name is made available to the interrupt task. Finally, the Cth virtual processor is made to run. After having carried out the actions implied by the interrupt request, the interrupt

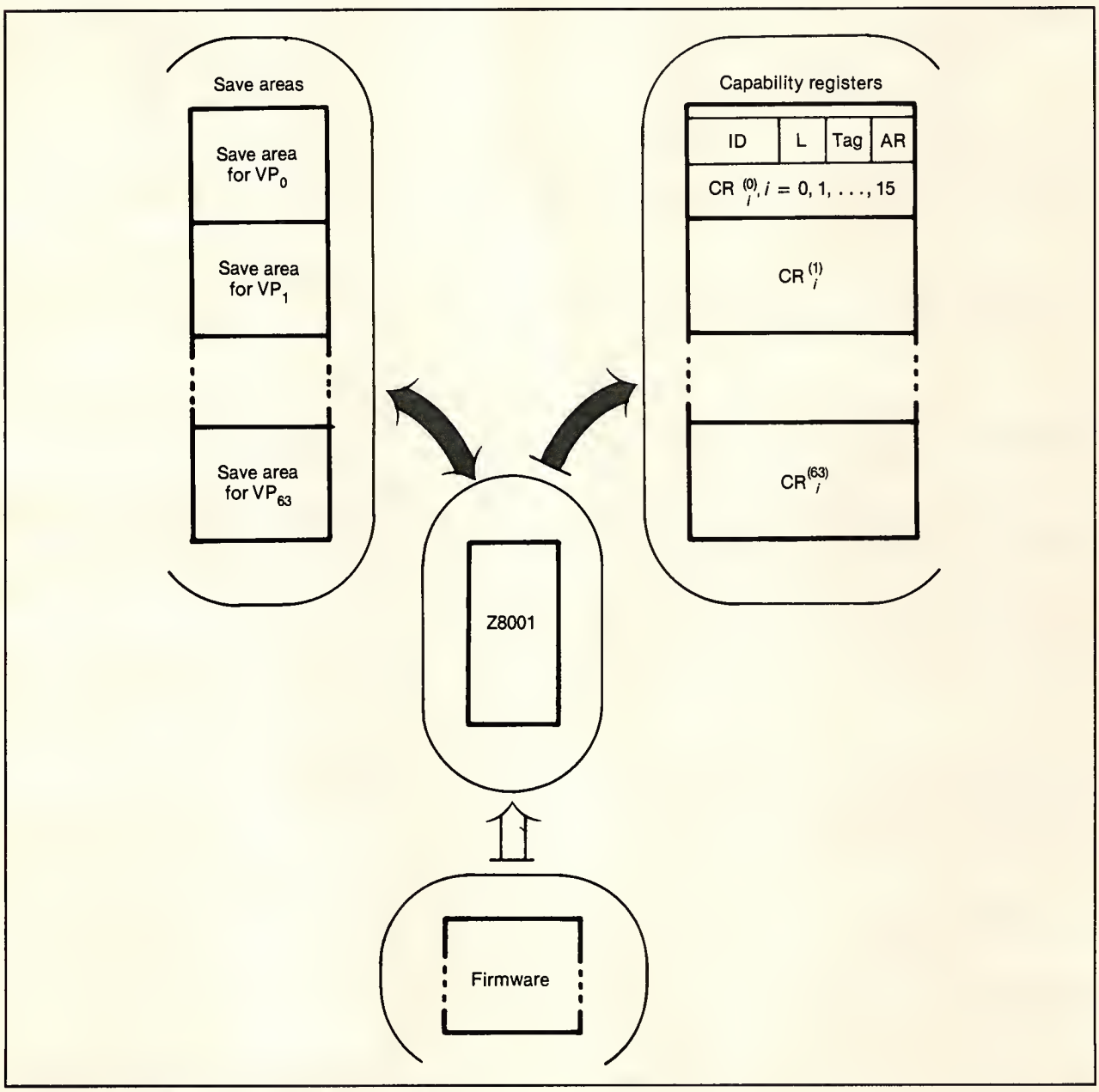


Figure 4. Logic diagram of the central processing unit.

task may suspend itself by means of the parameterless special instruction INTERRUPT_HANDLED. Execution of this instruction consists essentially of accessing the interrupt stack and popping the name of the virtual processor previously interrupted. This virtual processor is then made to run.

The intelligent memory device. The IMD is designed to support memory management. The CPU transmits the memory management instructions to the IMD by means of a memory-mapped communication channel. This channel consists of a few data spaces, all included in the same bootstrap family. The IMD contains the hardware and software resources it needs to execute memory management instructions (and, in particular, to perform family swapping) autonomously and in parallel with the operations of the CPU. Suppose that one such instruction is issued by a task *t* attached to the running virtual processor. The CPU only needs to transmit that instruction to the IMD and then switch to run a different virtual processor. The IMD generates an interrupt request upon completion of the execution of the instruction. The interrupt task honoring this request then returns control to *t*.

A block diagram of the actual configuration of the IMD is shown in Figure 5. A Z8001-based computer

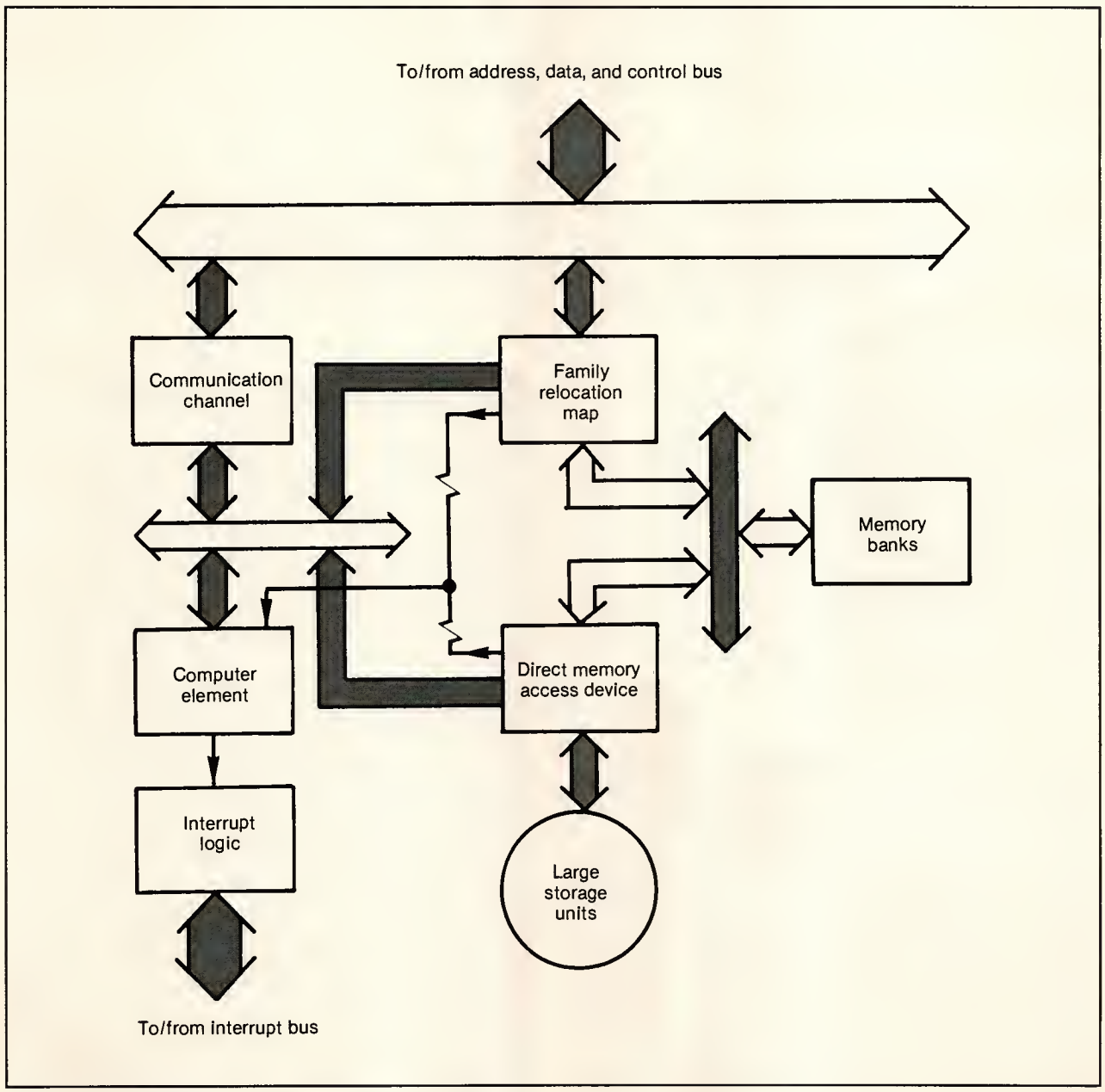


Figure 5. Block diagram of the intelligent memory device.

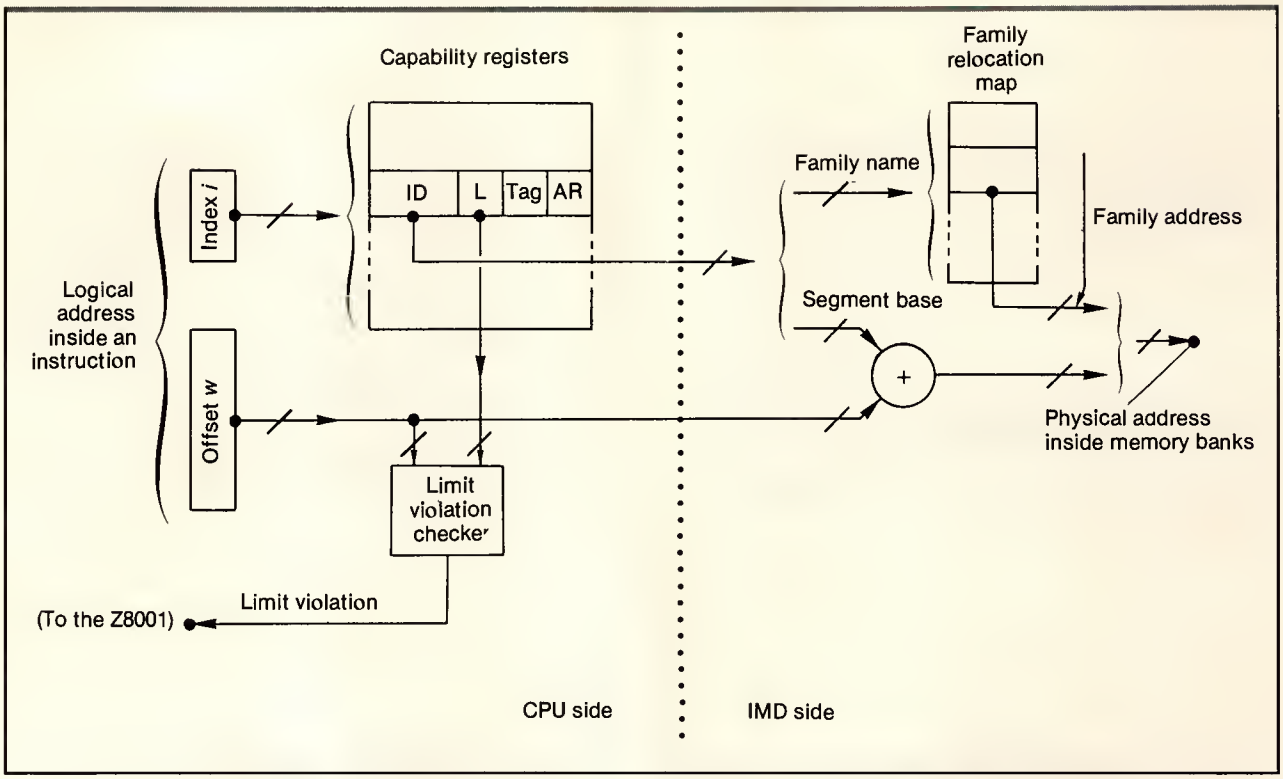


Figure 6. Translation of a logical address into a physical address in the memory banks. As a consequence of the way in which objects are generated (see Figure 2), the name of the family of the object being referenced is specified by the most significant bits of the identifier of that object. Moreover, the least significant bits represent the base of the segment implementing that object in the memory area in which the family is actually stored.

element inspects the communication channel by busy-waiting. After ascertaining the availability of an instruction from the CPU, the computer element executes the software routines implementing that instruction. (These are contained in the read-only memories inside the computer element itself.) If the need arises for a swapping action, the computer element simply activates a direct memory access device, which then performs the actual transfer of information between the large storage units and the random-access memory banks. This leaves the computer element free to start execution of another instruction available in the communication channel.

Address translation. Figure 6 shows how a logical address (as specified in a machine instruction) is translated into a physical address in the memory banks. As stated previously, a logical address consists of the name *i* of a capability register and an offset *w*. A limit violation checker compares the contents of the length field of the capability register with the offset and eventually generates a limit violation (actually, a segment trap) to the Z8001. Since this is not the case, the name of the family of the object referenced by the capability register is used to access an associative map, the family relocation map. This map is contained in the IMD. In this way, the address of the portion of the main memory reserved for that family is obtained. The base of the segment implementing the object inside this memory portion is then added to the offset and paired with the address of the family to finally obtain the physical address in the main memory.

Conceptually, the family relocation map should have one entry for each family in the main memory. Because technological constraints forbid fast associative memories of such depth, the associative behavior of the map is emulated as follows. A hash table, called the family relocation table, is implemented via software in the read/write memories of the computer element inside the IMD. The family relocation map is a buffer in which recently used family names are mapped into the corresponding memory addresses. Each entry of the family relocation map consists of a *tag field* and an *address field*. The least significant bits of the family name select a specific entry of the map. The most significant bits are compared with the contents of the tag field of this entry. If a match is found, the address translation is successful and the contents of the address field actually represent the starting address of the family in the main memory. Otherwise, an interrupt request is generated to the computer element. As a consequence of this interrupt request, the computer element performs a hash search in the family relocation table and loads the missing information into the family relocation map. Note that a failure in this hash search means that the family is not open; in this case, the access attempt to the main memory is aborted and an interrupt request is sent to the CPU for notification of the failure.

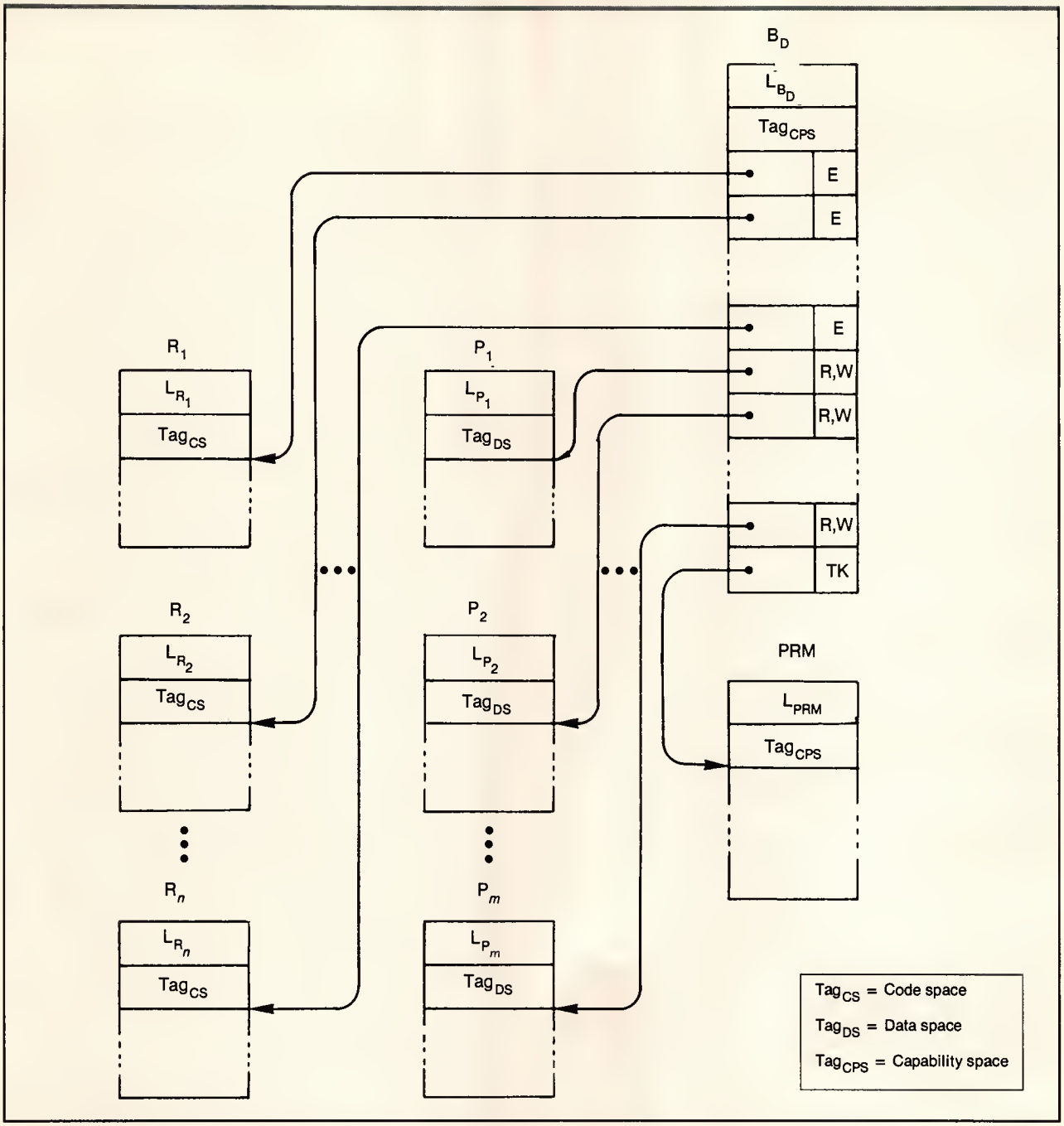


Figure 7. Configuration of a domain D implementing an object O of the abstract data type T.

Evaluation of the architecture

We faced several major decisions in designing our architecture. In particular, we had to devise strategies to ensure that it actually addressed the semantic gap problem.

Capability-based addressing. The major advantage ensuing from a capability scheme for memory addressing is effective runtime support for the implementation of abstract-type objects.

User-defined data abstraction. The importance of user-defined data abstraction as a major step toward structured programming is now widely appreciated. Facilities for the definition of abstract data types are common features of modern high-level programming languages. However, in a conventional addressing environment, the encapsulation of an abstract object, as specified by the high-level source code, is lost after compilation into machine code. This is not the case in a capability environment,[5,6] in which a protection domain hides the implementation of the object even at runtime, throughout object life.

Let us refer, for instance, to an object O of the abstract data type T. The configuration of domain D implementing O is shown in Figure 7. Code spaces R_1,

R_2,..., R_n store the routine for the operations of type T. Data spaces P_1, P_2,..., P_m contain the local variables that form the internal representation of O. Capability space PRM is used for the transmission of the capabilities for both the input parameters and the output parameters of the operations. The capabilities for these objects are grouped together in the base B_D of D. To execute the ith operation, a subject S must first store the capabilities for the parameters into PRM. Then, it must activate domain D and start execution of the routine contained in R_i. This routine then uses the capabilities in B_D to read the value of the input parameters, access the internal representation, and store the results of the operation into the output parameters. At completion of execution, the routine returns control to S. In this way, the representation of O may be accessed by the environment of the object only through the operations of the type.

Control over distribution of access privileges. Control over the distribution of access privileges is at the single-object level. Granularity is, therefore, much finer than in traditional architectures, which define only a few protection states (supervisor and user states, for instance). Any classification of programs as system and user programs is given up. Each program has its own set of access rights, and this set is the smallest one allowing the program to carry out its job. (This is the *principle of least privilege.*[2]) This feature can be important for error confinement as well as for fault detection, recovery, and retry. Moreover, it can provide help in all phases of software development.

Object sharing. A task holding a capability for a given object is free to transfer that capability to another task. In this way, the latter gains access privileges to the object. No intervention of the operating system is required. Indeed, flexibility in dynamic sharing of objects was the original reason for the introduction of the concept of a capability.[3]

Dangling references. The identifier of a deleted object is never used again for another object. Any attempt to utilize a capability to access a deleted object produces an access violation. Unique identifiers therefore represent an effective solution to the problem of dangling references.

Tagged memory. Every capability architecture must somehow prevent unauthorized accesses to the internal structure of capabilities.[7] Indeed, alteration of the contents of a capability may jeopardize the integrity of the whole protection system. A simple solution to this problem is to enforce separation of capabilities from data. This separation may be achieved by codifying the types of the entities contained inside an object in all the capabilities referencing that object, by means of different configurations of the access right fields. For instance, a capability for a capability space will never specify the access right WRITE, which makes it possible to freely modify the contents of that capability space.

A different approach consists of adding a one-bit tag to each memory cell; this bit specifies whether that cell contains a capability or not.[2] This approach, however, relies on specialized memory banks able to store cell tags. Moreover, whenever a portion of the main memory is swapped from/to the bulk memory, the relevant tag information needs to be swapped too, and this can be carried out efficiently only by ad hoc hardware. These disadvantages make the tag approach worth adopting only if it is used throughout the architecture, not only to separate capabilities from data but also to mark the other object types supported by the instruction set.[8] Indeed, if the specification of the type of an object is included in the object itself, it is possible to check at runtime whether the operations applied to that object are congruent with the object type. This may be useful, for instance, for detecting erroneous read accesses to uninitialized objects and for facilitating program debugging.

Our aim has been to enjoy all the advantages of such a type-safe environment but avoid the problems created by tag storage.[9] To this end, we have included the specification of the type of each object in a tag field within the segment implementing the object itself. This approach does not imply any specialized technique for information swapping. Moreover, with respect to the approaches mentioned above, it has the advantage of saving memory space. Indeed, with this approach a type specification has to be given only once for each object and does not have to be replicated in every capability for that object, or even in all the memory cells in which the object is stored.

Support for multitasking. A considerable drawback of capability-based systems is their need to convert a capability into the appropriate physical address upon each access to an object in memory. Most architectures partially solve this problem by including capability registers: for an object to be accessed, a capability for that object must be loaded into one such register. The contents of capability registers can be managed by ad hoc instructions or loaded autonomously by the microcode.[7] Visible capability registers have a potential for greater effectiveness. A drawback, however, is that a great deal of information must be saved at each task switch. This problem is even more serious if for kernel components such as interrupt tasks we wish to maintain the same degree of separation of privileges existing for user tasks.

These considerations convinced us of the need to make the CPU able to support several virtual processors. Indeed, as long as a task is attached to a virtual processor, that task can be run by issuing a single machine instruction, without incurring the time over-

head mentioned above.

The dualism of tasks and virtual processors allows us to create more tasks than the number of virtual processors actually implemented by the hardware. When a task T becomes ready and a virtual processor is available, a scheduler attaches T to this virtual processor. If a virtual processor is not available, the scheduler detaches a task T′ blocked with a low priority and attaches T to the virtual processor freed in this way. By working in this way, the scheduler will never detach a high-priority task. This is true, in particular, for interrupt tasks, each of which is attached permanently to a specific virtual processor.

Single-level store. A program written in a high-level programming language supporting the decomposition of programs into modules includes a great deal of information concerning memory usage. For instance, each module includes a visibility list specifying the names of the other modules that module can access. However, traditionally no such information is utilized for memory management. Instead, the execution of the machine code is inspected at runtime by an independently developed memory management system. This system tries to rebuild its own idea of the memory requirement of the running task by utilizing its own model of task behavior.[10] We believe that a better approach is

- to have the architecture include a set of memory management instructions that allow the running task to supply information concerning its own future need of memory resources, and
- to have the compiler generate memory management instructions in the object code that are coherent with the specifications of memory usage in the source program.

As a matter of fact, the instructions for memory management we have included in our architecture allow tasks to move families of objects through a two-layer physical memory hierarchy consisting of a fast-access main memory and a disk-based bulk memory. By doing so, these instructions implement the concept of a single-level store.[1] The salient features of this approach are discussed below.

Object life. The life period of an object is independent from that of the task creating that object. The object is deleted only when its own family is removed. Arrays provide not only for storage of short-term, small-sized objects in main memory but also for the permanent storage of large amounts of data. (In the latter case, files would be the traditional choice.) The greater homogeneity that ensues leads to generality in programs.[1] For instance, we only need a single routine even if the size of its parameters is such as not to allow us to store them in main memory. (An ad hoc I/O routine would have to be added in a conventional memory environment.)

Object size. No lower limit is imposed by the architecture on average object size. This feature is essential if we want to fully exploit the salient characteristics of an object-oriented organization.[11] A single table, the family relocation table, allows us to carry out the translation of the identifier of a given object into the address of that object in the physical memory. The number of entries in this table is equal to the number of families that may reside in the main memory at the same time. Each family has a capacity of 2_{13} bytes; the memory requirement for storage of the family relocation table is, therefore, quite low. Moreover, the dimension of families is independent of the average memory requirements for storage of a single segment. It follows that object size can be kept as small as desired, up to a lower limit of a few bytes for small-sized object types, without congesting I/O devices with a lot of swapping activities that each involve a small amount of data.

On the other hand, the fixed dimension of families implies an upper limit to object size. This drawback, however, is easily obviated. A large object unsuitable for storage in a single family is partitioned into a number of smaller objects, and these are allocated to different families. These smaller objects are included in the same domain. Suppose we activate this domain to carry out a specific operation on the composite object. We need to open only those families that actually contain components required for the actions involved in the operation.

Fragmentation. The size of a family is 2^{13} bytes. In the organization of a conventional system, swapping units of such an unusually large size would be likely to raise great fragmentation problems. In our approach, the compiler has control over the creation of objects inside families. As a result, fragmentation can be kept to a minimum.

The RISC approach. At present, the debate concerning the supposed advantages of reduced-instruction-set computer (RISC) architectures over complex-instruction-set computer (CISC) architectures seems far from resolution.[12,13] The benefits claimed for RISCs include higher code density, a consequence of their shorter instruction formats. However, several RISC instructions are often needed to express the actions contained in a single CISC instruction. RISCs can have smaller microprograms, leaving a larger chip area that can be profitably used for features such as on-chip caches and pipelining. However, it is probably far easier to enhance the performance of high-cost functions such as floating-point operations by ad hoc logic in a CISC architecture than it is in a RISC one.

The well-known fact that a few instruction opcodes cover most instruction executions suggests that a RISC design can be profitably tailored to a specific programming language. But it may well be difficult to maintain

the same advantages across a multiplicity of languages, since the sets of highly used instructions in those languages may differ substantially. (For instance, the Inmos Transputer RISC architecture is designed to support a specific programming language, Occam.) In compiler writing, an orthogonal instruction set simplifies code generation, and a hardware-supported high-level operation on a data type is often translated into a single instruction. However, a complex instruction that does not behave exactly as the compiler writer desires will probably never be used and, therefore, represents a useless complication of the architecture.

In our opinion, the strongest arguments in favor of RISCs are their shorter design time and improved testability. Design of RISCs is faster because of the comparative simplicity of their architecture, and their testability is enhanced by the small size of their microprograms. The result is not only reduced development costs but higher quality implementations, since designers can more easily use the latest technology. In addition, they can employ novel concepts in important aspects of computer organization such as the specification of operating systems and programming languages.

In our experiments, we added new instructions to the complex instruction set of the Z8001 (see box). At the hardware level, our CPU design effort involved implementing the ad hoc logic needed to support the activity of the microprocessor. At the software level, we had to write the routines emulating the special instructions. These routines required 680 Z8001 instructions (less than 2800 bytes of machine code). The routines relevant to the execution phase of the LOAD_CAPABILITY and STORE_CAPABILITY instructions, for instance, consist of 28 and 20 Z8001 machine instructions, respectively, and the memory requirement for their storage is 128 and 96 bytes. Eighty-two instructions (with a memory requirement of 328 bytes) were needed for the ACTIVATE_SUBJECT routine.

Therefore, in our experiments we have been successful in achieving a short design time and fast debugging. We were able to do this in a CISC architecture by using a conventional microprocessor in a novel way. Our complex instruction set is not simply the result of adding powerful instructions to a conventional von Neumann organization. We used special instructions and the segmented memory scheme of the Z8001 to implement advanced architectural features such as capability-based addressing and tagged memory. The salient advantages of these features have been described already. We are convinced they are worth the introduction of further complexity into the architecture, even at the expense of possible CPU performance degradation. Even this degradation may be able to be dealt with by increasing parallelism in the operations, which is what we did in our architecture for the intelligent memory device.

Why we chose the Z8001

The segmented memory scheme of the Z8001 allowed us to utilize its standard instruction set unconventionally. An address generated by the Z8001 consists of two components: a segment number and a byte number. We brought the visibility of capability register names up to the assembly language level by simply mapping each of the 16 first segment numbers into the name of one such register. The ability to do this was the main reason we used the Z8001.

Of course, this feature is not essential. A linear address, as generated by most microprocessors, can be easily translated into a pair {capability register name, offset} by reserving the most significant bits of the address for specification of the register name. Indeed, the only true requirement for the microprocessor is that enough address lines be available. This precludes the use of a microprocessor with a small address space (e.g., 2^{16} bytes).

The high-level-language architecture approach. The arguments for and against high-level-language machines have been discussed in depth by Ditzel and Patterson.[14] We are convinced that adequate hardware support should be provided to critical kernel functions. On the other hand, we wished to avoid the performance penalties that ensue when an instruction set optimized for a specific high-level language is used to implement languages for other classes of applications (e.g., when Lisp or Cobol is implemented on a Pascal or C machine). Therefore, we did *not* use a high-level-language architecture for our computer organization. Users are in fact aware of the transformation of their program from the source language into machine language. This was a conscious design choice. We did not include special instructions implementing language-specific features such as interprocess communication and fault treatment, for example, even though such features are supported by the instruction set of a machine like the Intel iAPX 432, which is tailored to the Ada language.[15]

In fact, our architecture strongly supports the implementation of high-level languages providing modular decomposition of programs, user-defined data abstraction, and concurrency. We do not rely on an ad hoc software structure, but we hypothesize a compiler taking advantage of the supports provided by the architecture for access mode checking, memory management, and tasking.

CPU performance. The performance of the CPU will now be analyzed in terms of capability loading and storing, domain switching, and task switching.

Capability loading and storing. About 310 clock cycles are required to emulate the execution phase of the LOAD_CAPABILITY instruction. At present, the Z8001 microprocessor operates with clock frequencies of up to 10 MHz. Our prototype features a 6-MHz clock; it follows that the execution time for this instruction is about 52 microseconds. The STORE_CAPABILITY instruction takes 220 clock cycles, or 37 microseconds. (The main reason for the different time performance of the two instructions is that execution of the LOAD_CAPABILITY instruction accesses the segment referenced by the capability involved. This access is required to convert that capability into a long capability.) Despite the very limited information transfer and data processing involved in a capability load and store, it can be seen that the execution times for these instructions are comparatively high. (The time taken to execute a Z8001 integer multiply instruction on 32-bit operands depends on the number of 1's in the multiplicand; it is 400 cycles on average. An integer divide instruction on operands of this size is carried out in about 725 cycles.)

Let us consider the routine relevant to the STORE_CAPABILITY instruction. The contents of the capability register involved are copied in memory by two Z8001 instructions in 54 cycles, whereas access right, tag, and limit checks are carried out in 166 cycles. Of course, a microprogram implementation could easily save most of these cycles by carrying out checks in parallel with memory accesses. Even at the emulation level, a mask register and a bound register (together with proper comparison logic) would allow us to reduce check times considerably. However, no such registers are at present included in our prototype. We have instead reduced the need for load and store capabilities by providing each virtual processor with 16 capability registers. These registers greatly help an optimizing compiler maintain a capability for a given object in the same register for as long as that object is likely to be referenced. This improves code performance in terms of both time and space.

Domain switching. Domain switches have always been considered the critical operation in capability environments. They have received considerable attention not only because of their high intrinsic cost but also because of their widespread use. Poor domain switch times cause the programmer to design his protection domains far larger than necessary for access control, object encapsulation, and object sharing. This occurs, in fact, in the Hydra operating system, in which more than 50 milliseconds are required to switch to the domain of the file system. This poor performance is of course a consequence of the fact that Hydra carries out domain switches entirely by software. On the other hand, the Cambridge CAP computer implements protection domain switching by microprogram, and the time it takes to switch from one domain to another and back again is 240 microseconds. In our architecture, the execution phase of the ACTIVATE_SUBJECT instruction is emulated in about 2400 cycles, or 600 microseconds with a 6-MHz clock frequency, whereas the DEACTIVATE_SUBJECT instruction takes about 1800 cycles, or 300 microseconds. Of course, timing comparisons between different machines give crude performance estimates, as they do not take into account important architectural factors such as word size. (The CAP is a 32-bit computer, for example.)

We can derive a rough best-case estimate of microprogram execution times by considering the number of memory accesses. Let us take the ACTIVATE_SUBJECT instruction as an example. Execution of this instruction copies the contents of a whole set of capability registers, of the program counter, and of the stack pointer onto the task stack. A long capability referencing a code segment is also loaded into a capability register. (Execution of the new subject will start in this code segment.) The size of a capability register is eight words, and a Z8001 read or write sequence takes three clock cycles. Therefore, a microprogram implementation of this instruction requires at least 400 cycles. An inspection of the routine emulating the ACTIVATE_SUBJECT instruction reveals several sources of time loss. More than 400 cycles are used for access right, tag, and limit checks. As mentioned previously, adequate hardware support would reduce this check time by at least one order of magnitude. Another time-consuming activity is the invalidation of capability registers, an operation needed to preserve the integrity of the objects in the domain being abandoned. We have actually made this activity faster by associating a flag—the VALID flag—with each capability register. A given capability register can be used to reference an object in memory only if its VALID flag is set. The flag is actually asserted when a capability is loaded into the register, and it is cleared during execution of the ACTIVATE_SUBJECT and DEACTIVATE_SUBJECT instructions. However, we had to implement the capability registers by means of random-access memory chips, and this is by far the greatest drawback to good time performance. In fact, with this approach, copying the contents of capability registers to/from the main storage becomes a memory-to-memory operation taking nine clock cycles for each word actually transferred.

The ACTIVATE_SUBJECT and DEACTIVATE_SUBJECT instructions involve almost the same number of information transfers with the main memory. Therefore, at microprogram level, these two instructions would have about the same cost. However, the DEACTIVATE_SUBJECT instruction involves neither

access right checks nor capability register invalidation. (It should be remembered that this instruction restores the contents of all registers with quantities popped from the task stack.) This is the reason for the much shorter emulation time of this instruction compared to that of the ACTIVATE_SUBJECT instruction.

Most modern high-level languages allow programs to be decomposed into concurrent, cooperating tasks to solve problems of large dimensions. In this approach, programmers are encouraged to enforce privilege separation by providing a task for each protection domain rather than by causing a single task to switch between many different domains. If this is done, domain switch times become less important than task switch times.

Task switching. We have taken great care to enhance efficiency in task switching. In fact, the multiple sets of capability registers permit us to perform a task switch by issuing a single instruction—i.e., a RUN_VP instruction—that takes 940 clock cycles. This is nearly half the time required by a software implementation of the same kernel function in an environment with a single set of capability registers. Furthermore, in such an environment, even a microprogram implementation of the RUN_VP instruction would take more than 900 cycles. Indeed, the state of a task includes the whole content of the capability registers, the general registers, and the status registers, and this content (150 words) must be transferred twice at each task switch.

Of course, task switching times would be enhanced by at least one order of magnitude in an architecture featuring multiple on-chip sets of both capability registers and general registers. We feel that this may well be a suitable way to use silicon area for maximized performance.

Nine-hundred clock cycles are required to carry out the execution phase of both the ATTACH_TASK and the DETACH_TASK instructions. However, since 64 sets of capability registers are available in our architecture, we seldom need to issue these instructions. In fact, after a task has been attached to a given virtual processor, it will probably never be detached before termination.

Pipelined execution of memory management instructions. Each memory management instruction is processed by a two-stage pipeline. Control is switched from the running task as soon as it issues one of these instructions. This prevents the first stage of the pipeline (i.e., the CPU) from filling up (unless the improbable situation of all the ready tasks needing a memory management activity occurs). Furthermore, because a task is supposed to enable a family only when it actually needs to access an object in that family, the pipe never needs to be emptied and performance degradation is avoided. As far as timing is concerned, CPU activity is not slowed down. This is essentially a consequence of the structure of the mechanism for address calculation. In particular, the family relocation map has been implemented so that memory cycle times are not extended. Of course, the CPU must be forced into a wait state if a miss occurs in the map when the contents of the map are being updated by the computer element inside the IMD.

We must point out that the hardware support we have provided in the IMD for low-level memory management activities is *not* a salient aspect of our architecture but only a feature of our particular implementation of it. The memory management instructions could well be implemented by software routines and executed by the CPU. Of course, this would result in a loss of parallelism in the operation.

The most important choice we made in designing our architecture was the one to use a capability-based addressing scheme. This decision dates back to the earliest stages of our research. It was dictated mainly by our previous studies on the semantic gap and, in particular, on the implementation of abstract object types.[5,6] Other aspects of our architecture benefited from the decision, too. The single-level store, for instance, had a positive impact on garbage collection and allowed us to take advantage of the modularized structure of programs, even though we introduced this approach to memory management mainly as a solution to a serious drawback of capability organizations, the cost (in both processing time and memory requirements) of mapping object identifiers into physical addresses in the main memory. Similarly, we saw tagged memory segments mainly as a means of segregating capabilities from data, although this tagging scheme influenced the whole structure of the software; in particular, it caused a vertical migration of object encapsulation down to the hardware level.

As far as implementing the architecture was concerned, our most important decision was to use an off-the-shelf microprocessor for the CPU. Another basic choice was the one to support the kernel functions of memory management by means of ad hoc processing power. Neither of these decisions had a major impact on the architecture. We hope our experiment will encourage the use of conventional microprocessors for the implementation of novel architectures and advanced machine organizations.

Acknowledgments

The work on which this article is based was supported in part by the Italian Ministry of Education and in part under a research contract between Selenia, Industrie Elettroniche Associate, SpA, Rome, Italy, and the Consiglio Nazionale delle Ricerche (National Council of Research), Pisa, Italy.

References

1. G.J. Myers, *Advances in Computer Architecture*, 2nd ed., John Wiley & Sons, New York, 1982.
2. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, Vol. 63, No. 9, 1975, pp. 1278-1308.
3. R.S. Fabry, "Capability-Based Addressing," *Comm. ACM*, Vol. 17, No. 7, 1974, pp. 403-412.
4. *Z8000 CPU Technical Manual*, Zilog, Inc., Cupertino, Calif., 1980.
5. P. Corsini, G. Frosini, and L. Lopriore, "The Implementation of Abstract Objects in a Capability Based Addressing Architecture," *Computer J.*, Vol. 27, No. 2, 1984, pp. 127-134.
6. P. Corsini, G. Frosini, and L. Lopriore, "Distributing and Revoking Access Authorizations on Abstract Objects: A Capability Approach," *Software—Practice & Experience*, Vol. 14, No. 10, 1984, pp. 931-943.
7. H.M. Levy, *Capability-Based Computer Systems*, Digital Press, Maynard, Mass., 1984.
8. E.A. Feustel, "On the Advantages of Tagged Architecture," *IEEE Trans. Computers*, Vol. C-22, No. 7, 1973, pp. 644-656.
9. L. Lopriore, "Capability Based Tagged Architectures," *IEEE Trans. Computers*, Vol. C-33, No. 9, 1984, pp. 786-803.
10. P.J. Denning, "Virtual Memory," *Computing Surveys*, Vol. 2, No. 3, 1970, pp. 153-189.
11. E.F. Gehringer, *Capability Architectures and Small Objects*, UMI Research Press, 1982.
12. D.A. Patterson and D.R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News*, Vol. 8, No. 6, 1980, pp. 25-33.
13. D.W. Clark and W.D. Strecker, "Comments on 'The Case for the Reduced Instruction Set Computer' by Patterson and Ditzel," *Computer Architecture News*, Vol. 8, No. 6, 1980, pp. 34-38.
14. D.R. Ditzel and D.A. Patterson, "Retrospective on High-Level Language Computer Architecture," *Proc. 7th Ann. Int'l Symp. Computer Architecture*, Computer Society Press, Washington, D.C., 1980, pp. 97-104.
15. E.I. Organick, *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, 1983.

Lanfranco Lopriore is a researcher at the Istituto di Elaborazione della Informazione of the Consiglio Nazionale delle Ricerche, Pisa, Italy. He has researched memory structures in capability environments, capability-based tagged architectures, and cache memories. He is currently working on hardware tools for program debugging and program performance evaluation.

Lopriore received the Dott. Ing. degree in electronic engineering in 1978 and an advanced degree in computer science in 1980, both cum laude, from the University of Pisa.

Questions about this article can be directed to Lopriore at the Istituto di Elaborazione della Informazione, Consiglio Nazionale delle Ricerche, Via Santa Maria 46, 56100 Pisa, Italy.

Paolo Corsini is a professor of digital computers with the Engineering Faculty of the University of Pisa. His research interests include multi-microprocessor systems and computer architecture. He received the Dott. Ing. degree in electronic engineering, cum laude, from the University of Pisa in 1969.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 156 Medium 157 Low 158

Improved Control Acquisition Scheme for the IEEE 896 Futurebus

D. Matthew Taub
IBM United Kingdom Laboratories Ltd.

An added preemption facility clearly improves earlier schemes for implementing this backplane bus used with 32-bit microprocessors.

The performance of a multi-microprocessor system depends to a great extent on the facilities provided by the backplane bus through which the microprocessors are interconnected. For the new generation of 32-bit microprocessors, several buses have been introduced or proposed. These include the Motorola VMEbus, the Texas Instruments Nubus, the Intel Multibus II, the IEEE 960 Fastbus, and the IEEE 896 Futurebus. There has been much discussion of their relative merits.[1-5]

The most ambitious appears to be Futurebus. It is asynchronous, it requires no centralized control, it supports fault-tolerant and cache-based architectures, and it allows modules to be added or removed while the system is running (live insertion and withdrawal). Edwards and Peyton-Jones discussed the importance of these facilities.[6,7]

The technical details of Futurebus appeared in several articles published in *IEEE Micro* in August 1984. These articles described the scheme as it existed in Draft 6.2 of the Specification. Among them was an article I wrote on the arbitration and control acquisition arrangements.[8] At the end I pointed out that the article did not necessarily represent the final word and that further improvements might yet be made. This has indeed happened, and the present article explains what these improvements are and what advantages they give.

The main improvement is the introduction of a preemption scheme, which ensures that modules urgently needing the bus are not kept waiting any longer than necessary. We also corrected the calculations concerned with the settling time of the arbitration circuits and improved the scheme whereby a module newly live-inserted into the bus establishes synchronism with the modules already working.

Main features of the Draft 6.2 scheme

In this early scheme, modules requesting control of the bus signalled their request over bus line AC to the module currently in control, known as the current master or, more briefly, the master. The master module responded at a suitable time by starting the control acquisition procedure. This procedure consisted of a sequence of six operations.

During operation 1 modules decided whether or not they were competing for the bus on this occasion, and during operation 2 arbitration took place using the well-known parallel scheme common to IEEE 696 and several other buses. The purpose of operation 3 was to check for arbitration errors, and assuming none were found, operation 4 provided the master

0272-1732/87/0600-0052$01.00 © 1987 IEEE

with time to finish its data transaction. In operation 5 modules carried out the various tasks needed when control of the bus was transferred; in operation 6 modules registered the identity of the new master.

During the above procedure, the operations in the various modules were kept in synchronism using the three bus lines AP, AQ, and AR. The technique is very similar to that used for the data strobe and acknowledge signals in Trimosbus.[9]

Another feature of the scheme was the division of the modules into two classes, a priority class whose members competed for the bus whenever they needed it, and a fairness class whose members, once having had control of the bus, were barred from competing for it a second time until no unfulfilled bus requests remained. Removal of the bar took place during a three-operation procedure, initiated as with the main procedure, by the current master.

A deficiency of the Draft 6.2 scheme

Suppose that one or more bus requests arose while the master was carrying out its data transactions. The ideal time for the master to start the control acquisition procedure would be shortly before its transactions were complete, so that the end of the transactions coincided as nearly as possible with the end of operation 3 in the control acquisition procedure. But estimation of this time would have been difficult and would have required extra software. What was much more likely to happen, therefore, was that current masters would either have delayed starting the acquisition procedure until their transactions were finished, which would have reduced the bus throughput, or they would have started it as soon as a bus request was received.

In the latter case the next master, known as the master elect, could have been chosen early in the current master's tenure of the bus, resulting in its spending a substantial part of its total tenure period in operation 4. A problem could arise if, during that period, a priority-class module developed an urgent need for the bus; there was no way for it to signal its request to the master. It would have had to wait not only until the master had finished with the bus but until the master-elect had finished as well. The changes to the control acquisition scheme, described below, overcome this problem by allowing the priority-class module to displace the master-elect, an arrangement known as preemption.

Unchanged features

Before describing the changes, it is worth drawing attention to the features that remain the same. They are:

• *The arbitration method.* This still works exactly as described in the 1984 article[8]; that is to say, each module is assigned its own 7-bit arbitration number.

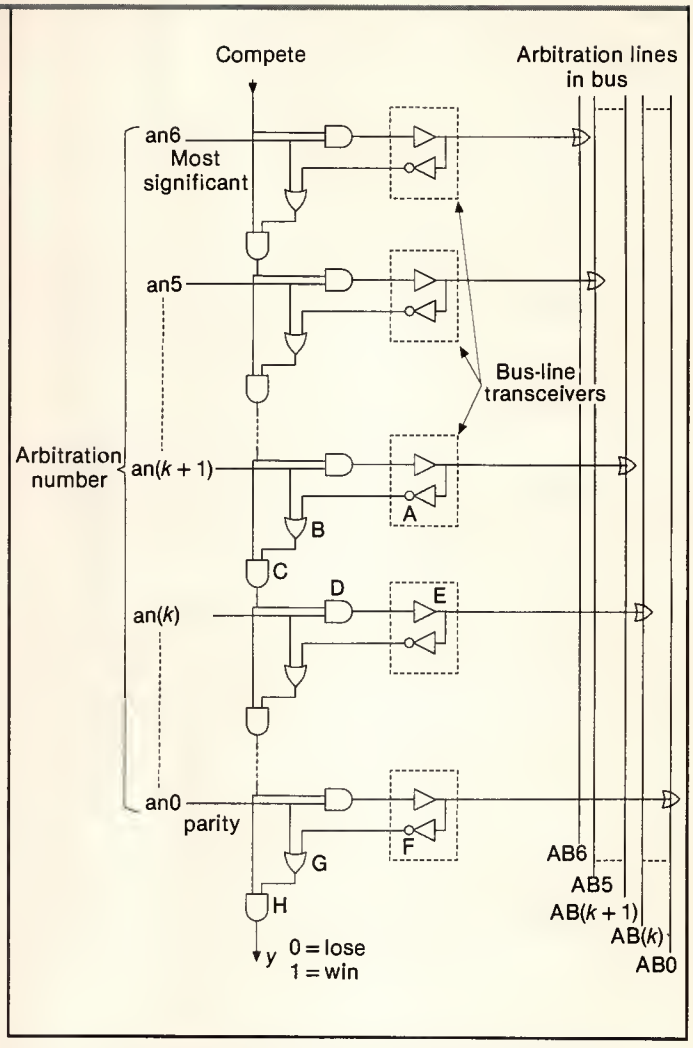


Figure 1. Arbitration logic. This is a pure logic diagram showing the relationship between Boolean variables rather than between electrical levels. The parity bit an0 is chosen to give odd parity over an6 to an0.

During arbitration the module applies this number through open-collector stages to seven corresponding bus lines, AB6 to AB0, all of them active-low. (As in the 1984 version, the asterisks commonly used in IEEE documents to distinguish active-low from active-high lines have been omitted in the interests of simplicity. The lines now designated AB were previously designated AN. The change was made to avoid an inconsistency in the specification.)

At the same time the module monitors the lines, and if it is applying a 0 to a line but senses that the line is carrying a 1 (applied by another module), then for as long as this condition persists, it ceases applying all its arbitration-number digits of lower significance. The result is that when the circuit has settled to a steady state, the AB lines carry the highest of the arbitration numbers applied by the competitors, and the module with this arbitration number is the winner. A suitable logic circuit is shown in Figure 1.

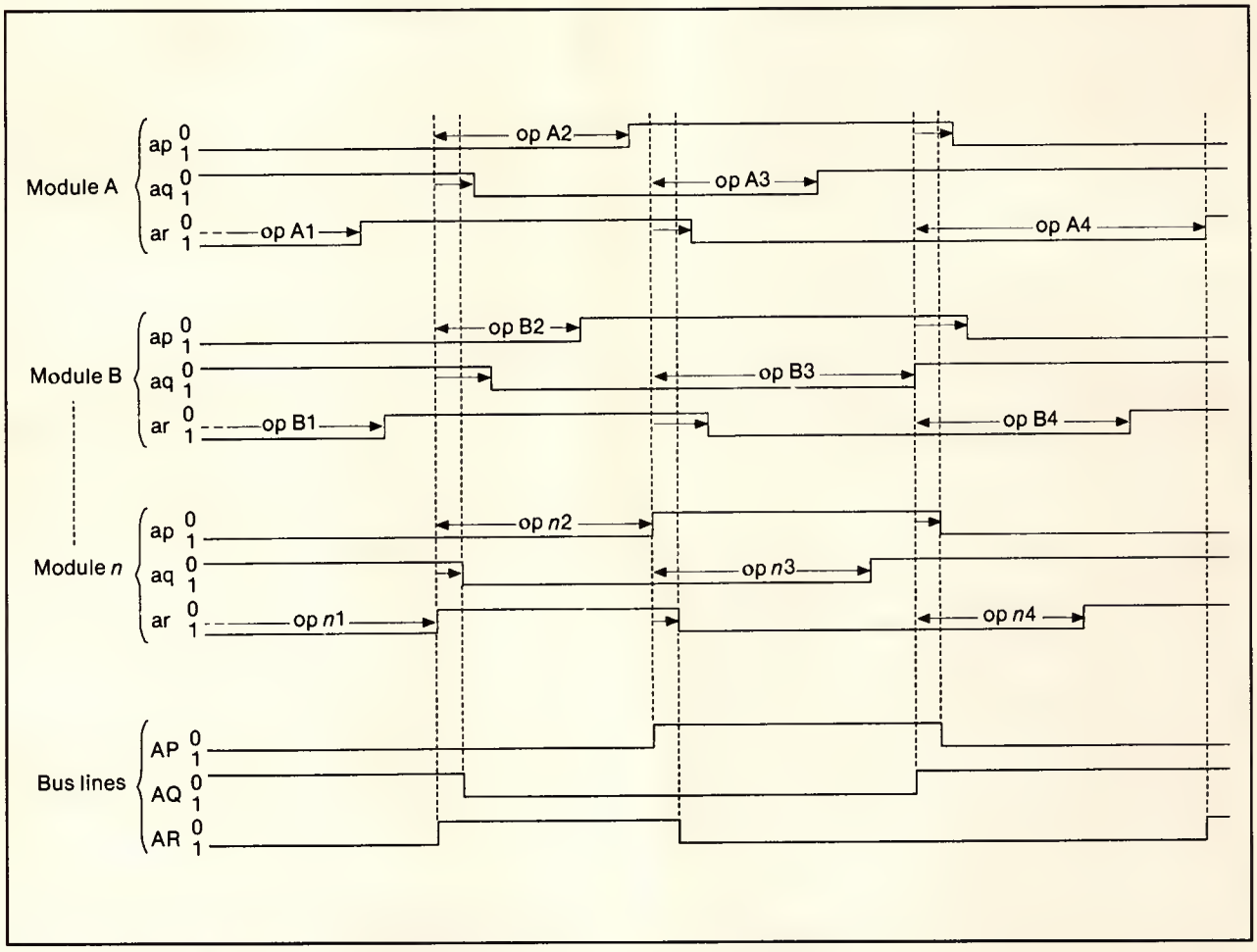


Figure 2. Module signals and resulting bus-line signals for achieving synchronization.

• *The synchronization method.* This method still follows the same principles as before, but there are differences of detail; a complete description appears later. Also, some of the actions in the procedure have been transferred to different-numbered operations; for example, error checking now takes place in operation 5 and hand-over tasks, in operation 6.

• *Fairness and priority classes.* These two classes, and the rules governing the circumstances under which their members may compete for the bus, are unchanged.

Changes in the control acquisition procedure

In the following explanation we say a signal is asserted when it has the value binary 1, and released when it has the value binary 0. As with the AB lines all the bus lines are driven from open-collector stages and are active-low; that is, they all carry out the wired-OR function, binary 1 being represented by the less-positive level. The variables that an individual module applies to the bus lines are denoted by lowercase letters and the bus lines themselves, by the corresponding capital letters.

Starting. Instead of being started by the current master, the control acquisition procedure is now started by any module requiring the bus; this includes modules that are barred by the fairness rule from taking part in arbitration. A module starts the procedure by asserting bus line AP.

Synchronization method. Rather than describing the synchronization method in terms of changes, I explain the method "from scratch" so it will be clearer, particularly for readers not too familiar with the earlier article. The objective, as before, is to keep all the modules synchronized throughout the acquisition

procedure; that is, to ensure that no module can start operation $(i+1)$ until all have finished operation i.

Basic method. Consider a group of modules of different speeds that have to carry out a sequence of operations in synchronism as indicated above. The operations form a loop, so that as soon as the slowest module completes the last operation in the sequence, all modules start the first operation over again, and so on ad infinitum. We start at the point in the loop where all the modules are engaged on the first operation, operation 1, but none have yet finished it. At this point, ap, aq, and ar in all the modules, and therefore the signals on bus lines AP, AQ, and AR, are 1, 0, 1 respectively, as shown in Figure 2. This means AP and AR are asserted and AQ is released. The way in which these signals change as the sequence progresses is as follows:

1) As soon as each participant completes its first operation, it releases its ar. Only when the slowest has done so will line AR be released. In the example shown, the slowest module is seen to be *n*.

2) All participants respond to the release of AR by asserting their aq and starting their second operation. As soon as they complete it, they release ap. The release of AP indicates that the slowest has finished. In this example it again happens to be *n*.

3) All participants respond to the release of AP by asserting ar and starting their third operation. When they have completed it, they release aq, and similarly, the release of AQ indicates that the slowest has finished (this time, module B).

4) All participants respond to the release of AQ by asserting ap and starting their fourth operation. On completing it, they release ar.

The process continues as above, the bus signals in operation 4 being the same as in operation 1, those in operation 5 being the same as in operation 2, and so on. Therefore, if the state of the bus lines at the beginning of the sequence is always to be the same, which in Futurebus is a requirement, the number of operations in the sequence has to be an integral multiple of 3. Table 1 summarizes the bus-line states as the sequence progresses.

A problem with wired-OR lines such as AP, AQ, and AR, is that the release of a line by one module while another is still holding it asserted can cause a glitch to appear on the line. This results from the change in the current-flow pattern. Futurebus solves the problem by taking the relevant bus-line receiver outputs through integrators and threshold circuits. These are designed so that the longest possible glitch or succession of glitches that can last for up to twice the end-to-end propagation time of the bus will not cause the threshold circuit to switch. The maximum propagation time in Futurebus is 12.5 ns, and so the integrator/threshold circuit combination must suppress glitches lasting for 25 ns or less.

Table 1.
Bus-line synchronization signals.

State	AP	AQ	AR
Op 1 in progress	1	0	1
Op 1 complete	1	0	0
Op 2 in progress	1	1	0
Op 2 complete	0	1	0
Op 3 in progress	0	1	1
Op 3 complete	0	0	1
Op 4 in progress (as Op 1)	1	0	1

Method as applied to Futurebus. The new Futurebus sychronization method has several features in addition to those described above. They are as follows:

- Sequences are of two possible lengths: a three-operation sequence for cancelling the arbitration bar in fairness modules and a six-operation sequence for normal arbitration and transfer of control. The decision as to which length of sequence is required is made shortly after the sequence starts (operation 2) and is stored internally in all participating modules.
- Strictly speaking, the number of separate operations needed in the longer sequence is only four. But this sequence is extended to the required figure of six by allowing the settling of the arbitration circuits to spread over operations 2, 3, and 4. This gives faster overall performance than does the alternative of introducing dummy operations (no-ops).
- A pause is introduced during operation 1 to wait for one or another of the potential masters to request control of the bus. This occurs as follows. When modules detect AQ switching to 0 and indicating that the last operation in the preceding sequence is finished, they do not automatically assert ap. The module or modules requiring the bus assert ap first, and the remainder follow suit only when they detect that AP is asserted. Until this happens, no module is permitted to signal the end of operation 1, that is, by releasing ar.
- A pause is introduced during operation 5 mainly to wait for the current master to finish its bus transactions. This occurs in similar fashion to the above. When AR switches to 0 indicating that all modules have completed operation 4, modules do not automatically assert aq. The first one to do so is either:

1) a requester, if there is no current master; (The absence of a current master will have been detected in operation 1 by all the AB lines being released; see

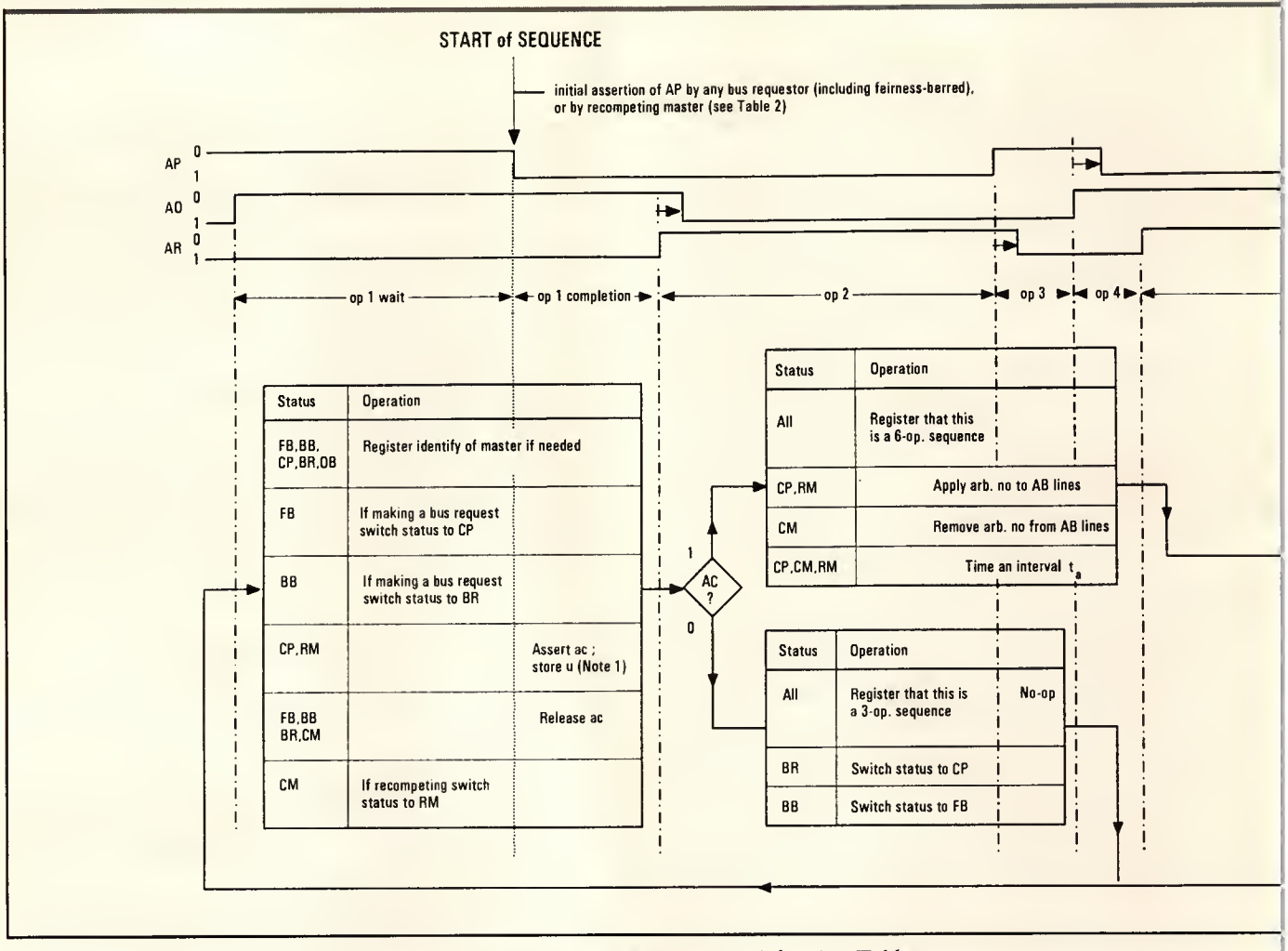


Figure 3. Control acquisition sequence. The two-letter status designations are defined in Table 2.

Note 1 in Figure 3.);

2) any potential master detecting an error in the arbitration that has just taken place;

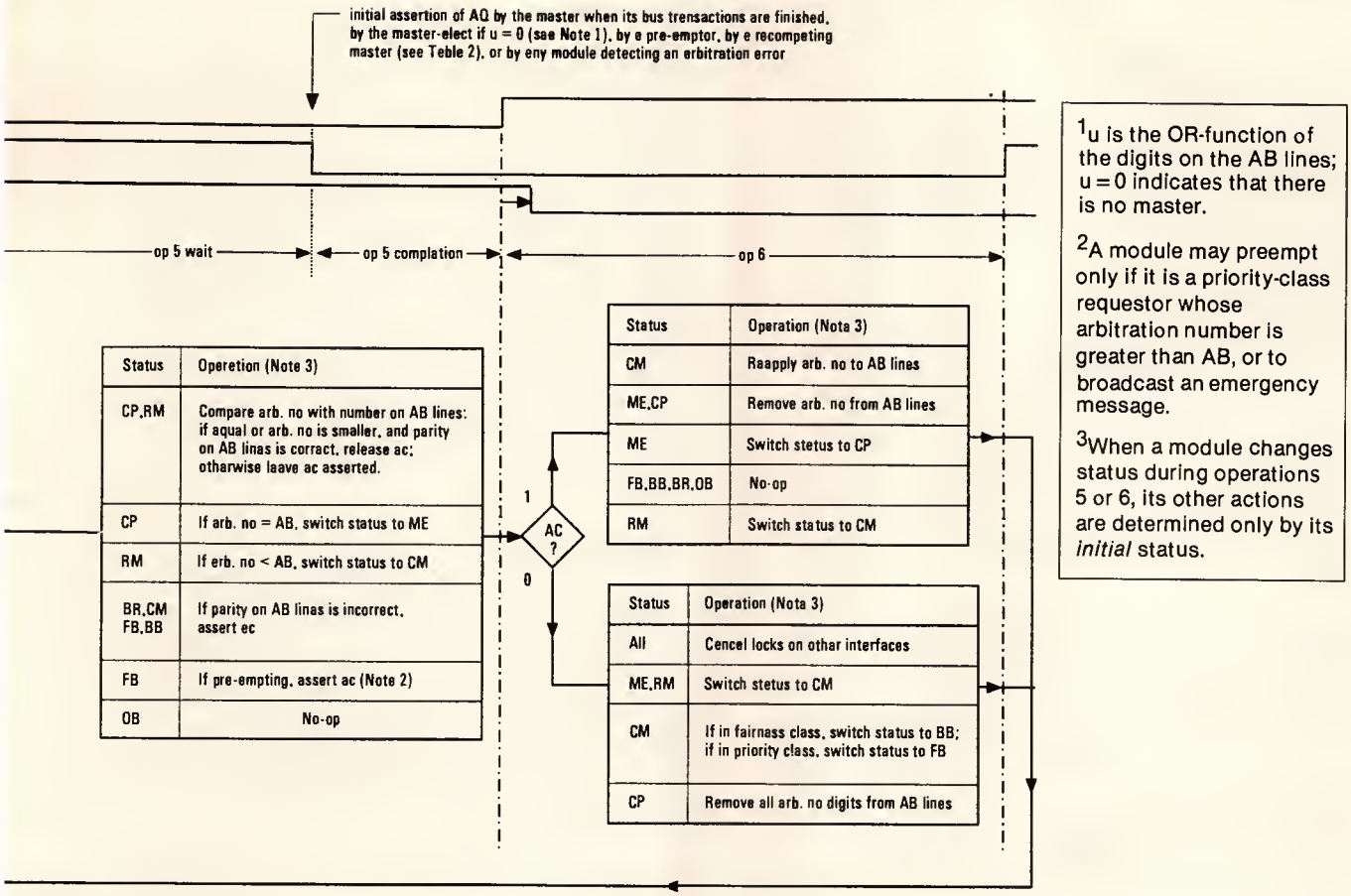
3) a priority-class module seeking to displace the master-elect (preemption); or

4) the current master, when it has finished its bus transactions. The remaining modules do not assert their aq until they detect AQ asserted, and only after this has happened are they permitted to release ap indicating that they have finished operation 5.

Operations 1 and 5 may therefore be thought of as being divided into two periods: a *wait* period that lasts until the appropriate variable, ap or aq respectively, is asserted, and a *completion* period that lasts from then until the required operation is over. Note that the wait period can be arbitrarily short; for instance, a bus request can be presented immediately after the preceding sequence finishes, possibly by a fairness module whose bar to arbitration has just been cancelled; a module that lost in the preceding arbitration; or a module that caused preemption to take place. In operation 5 the wait period can be arbitrarily short if there is no master.

Table 2. Module status.

Designation	Meaning	Definition
FB	Free bystander	A module not at the time barred under the fairness rule and not requesting control of the bus
BB	Barred bystander	A module barred by the fairness rule from taking part in arbitration and not requesting control of the bus
CP	Competitor	A module requesting control of the bus and free to take part in arbitration



[1]u is the OR-function of the digits on the AB lines; u = 0 indicates that there is no master.

[2]A module may preempt only if it is a priority-class requestor whose arbitration number is greater than AB, or to broadcast an emergency message.

[3]When a module changes status during operations 5 or 6, its other actions are determined only by its *initial* status.

Table 2 (cont'd.)

Desig-nation	Meaning	Definition
BR	Barred requestor	A module requesting control of the bus but barred by the fairness rule from taking part in arbitration
CM	Current master	The module currently in control of the bus
ME	Master-elect	The competitor that has won the immediately preceding arbitration but has not yet become master
OB	Observer	A module that never requires control of the bus but takes part in the control acquisition procedure to synchronize certain tasks with other modules[1]
RM	Recompeting master	A status assumed by the master to initiate a dummy control acquisition procedure[2]

[1] The tasks that an observer may have to carry out include: (a) registering the identity of the current master, (b) unlocking other interfaces on completion of the master's bus transactions, and (c) receiving emergency messages.

[2] The master can initiate a dummy control acquisition procedure if it finishes its data transactions before any other module requests control of the bus. The purpose is to prevent other interfaces in the slaves from remaining locked for any longer than necessary (p. 36[8]). Alternatively, the master can unlock its slaves using the data-transfer portion of the bus.

The new procedure

The flow diagram in Figure 3 presents details of the new control acquisition procedure; Table 2 gives the meaning of the various two-letter status designations. The main features of the procedure are described here.

Operation 1. Modules register the identity of the current master, that is, the number on the AB lines. Any module or modules requiring control of the bus assert ap to start the procedure, and if not barred under the fairness rule, assert ac.

Operations 2, 3, and 4. Competing modules engage in arbitration and time an interval t_a to allow the arbitration circuits to settle. The value of t_a is discussed later.

Operation 5. All potential masters check for arbitration errors. Modules finding an error or carrying out preemption assert ac to prevent bus mastership from being transferred and assert aq to restart the sequence. Otherwise, modules wait until the sequence is restarted by the current master after it has finished using the bus. Note that a module may preempt another only if it is in the priority class and if its arbitration number is higher than that of the master-elect, that is, higher than the number on the AB lines.

Operation 6. If AC = 0, indicating that control of the bus is to be transferred, the master-elect becomes the new master, and all modules cancel any locking operations imposed during the previous bus tenure. If AC = 1, the current master remains in control, and all the conditions that existed before the sequence started are reestablished.

Emergency messages

The preemption feature in the new scheme provides a powerful method of broadcasting emergency messages such as warning of an imminent power failure. One pays a price for this facility in that each possible emergency message reduces by one the number of priority-class modules that can be accommodated. But in practice this is unlikely to be a serious limitation. The method is as follows.

Suppose that a total of four different emergency messages is required. The four contiguous numbers at the top of the priority-class range of arbitration numbers represent these messages. The highest number, that is, 1111111, represents the most urgent message, and the lowest number, 1111001, represents the least urgent. (Note that the least significant bit is a parity bit giving odd parity overall).

If the system is in operation 5 waiting for the current master to finish its transactions, and one of the modules needs to send an emergency message, it causes preemption to take place as described above. As a result, the system reaches operation 1 with the current master still in control. The module sending the message then immediately starts a new procedure in which it enters arbitration using the appropriate emergency-message number instead of its normal arbitration number. Provided no other module is sending a message of higher urgency, the number appearing on the AB lines during the following operation 5 will be the number representing the message. All modules taking part are required to act on it. The module sending the message then causes preemption to take place a second time, returning the system to operation 1 with the current master still in control.

Correction to the expression for t_a

The length of time that the arbitration circuits take to settle depends on two types of delays. One is the propagation delay along the bus; a second type concerns delays through the logic circuits, bus-line transceivers, and antiglitch integrators in the competing modules and current master. An expression for the settling time under worst-case conditions t_{as} was derived in the 1984 article. For the general case of n arbitration lines, it is:

$$t_{as} = 4t_p + \underset{c}{\mathrm{Max}}\,(t_s + t_d) + \left[\sum_{k=0}^{n-2} \underset{c}{\mathrm{Max}}\, t_{e,k}\right] + \underset{c}{\mathrm{Max}}\, t_f \quad (1)$$

where t_p = maximum end-to-end propagation delay along the bus;

t_s = delay introduced by the integrator following the AR bus-line receiver;

t_d = delay between the release of AR at a module's terminal and the most significant digit of its arbitration number being applied to AB6, less the delay introduced by the AR integrator;

$t_{e,k}$ = delay between an externally produced change on a module's AB($k+1$) bus-line terminal and the resulting change on its AB(k) terminal, for example, in Figure 1 the sum of the delays through elements A, B, C, D, and E;

t_f = delay between an externally produced change on a module's AB0 terminal and the resulting change in its win/lose signal y in Figure 1, that is, the sum of the delays through elements F, G, and H; and

Max/c stands for the maximum value over the competitors and current master.

In fact Equation 1 cannot be used directly as a basis for modules to time the arbitration process (operations 2 through 4) because no module has information on the circuit delays in the others. Instead all the competitors and the current master introduce a delay t_a that depends only on its own circuit delays

and on values that are specified for the whole system. (This interval was previously designated t_2 because the delay was introduced during operation 2. It now extends through operations 2, 3, and 4, and so the old designation is no longer appropriate.) t_a thus varies from module to module. The expression for it has to be such that the highest value among the competitors and current master is never less than t_{as} as given by Equation 1. This guarantees that operation 5 cannot start until the settling process is complete.

t_a can be considered as the sum of three components, t_{a1}, t_{a2}, and t_{a3}. t_{a1} depends on the bus-propagation delay, t_{a2} on logic-circuit and bus-line transceiver delays, and t_{a3} on the integrator delay. In 1984 it was proved that suitable expressions for t_{a1} and t_{a2} are:

$$t_{a1} = 4t_p \qquad (2)$$

$$t_{a2} = (n+1)\,\mathrm{Max}\left[t_d, t_{e,k}(0 \le k \le n-2), t_f\right] \qquad (3)$$

where Max stands for the largest component among t_d, all the $t_{e,k}$ and t_f within the module in question.

However, the 1984 version needs to be corrected concerning t_{a3}. It gave t_{a3} the value $t_{s\,max}$, that is, the maximum delay that the integrator/threshold circuit in any module is allowed to introduce. In fact this is longer than necessary; the original reason for the term $t_{s\,max}$ and the correct term follows.

Original reasoning. Under the conditions present in Futurebus, the maximum duration of a glitch or glitches that a module can experience on any bus line preceding its genuine release is $2t_p$. A simple case in which a glitch of this duration occurs is when there are two modules at opposite ends of the bus, both holding a line asserted, and one releases it; the module that released it will experience a $2t_p$ glitch. The integrator/threshold-circuit combination included in every AP, AQ, and AR bus-line receiver rejects all pulses up to and including this value.

We are concerned with the delay t' that the integrator can introduce between the genuine release of a line and the module perceiving that release, that is, the interval until its threshold circuit switches.

Suppose that a glitch has brought the integrator output in a module to just below the threshold level, and that shortly afterwards, before the integrator has had time to reset appreciably, a genuine release of the bus line takes place. Then t' in that module will be negligibly small, whereas in another module not experiencing any glitch, t' may be as high as $t_{s\,max}$.

The features of the worst case conditions relevant to the present argument are shown in Figure 4 (derived from the 1984 Figure 2 on p. 33[8]). The master is situated at one end of the bus, and PM4 at the opposite end is one of the potential masters about to take part in arbitration. (The "4" in its designation is used simply to maintain consistency with the 1984 version.)

In the 1984 reasoning, PM4, the ultimate winner of the arbitration, was assumed to be the last module to complete operation 1, and so line AR at the master would not be released until t_p later. It was argued that in PM4 t' could be zero, while in the master it could be $t_{s\,max}$. And so, to allow for the latter's late start of arbitration, PM4 and therefore modules in general would have to make t_{a3} equal to $t_{s\,max}$.

In fact this argument is false. If PM4 is the last

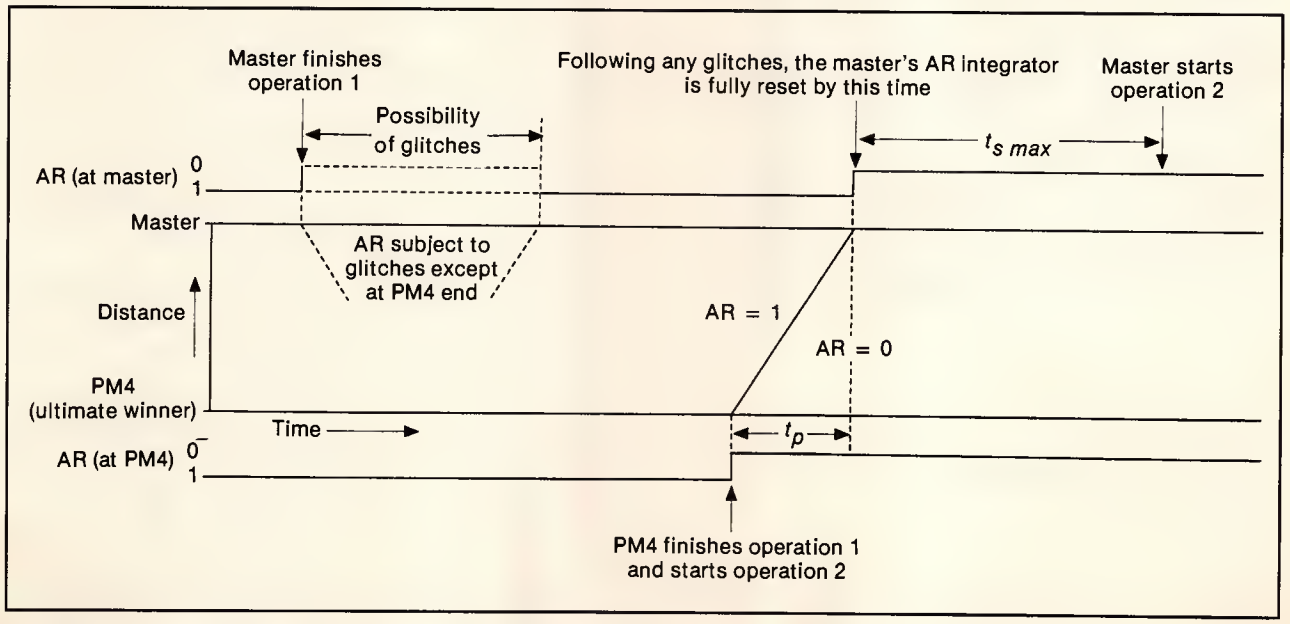


Figure 4. Waveforms and lattice diagram showing conditions wrongly assumed in the earlier calculation of t_a.

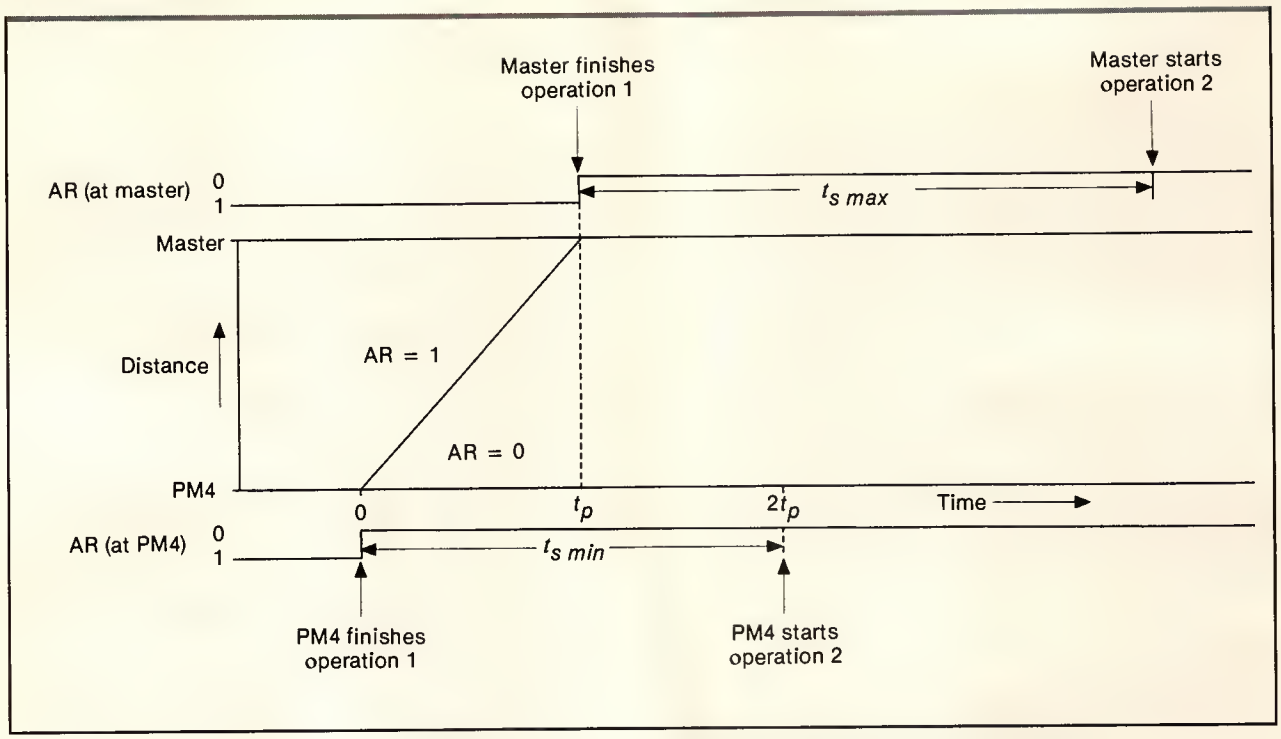


Figure 5. Waveforms and lattice diagram showing corrected conditions for worst effect of integrator delay.

module to finish operation 1, it cannot have experienced any glitches, and so its value of t' must be at least $t_{s\,min}$.

Conditions for the longest arbitration settling time. The arbitration settling time experienced by PM4 will be longest when PM4 starts the arbitration operation as early as possible and the master as late as possible. Figure 5 shows the circumstances under which this occurs; it deliberately ignores bus-line receiver delays because these have already been accounted for as part of t_{a2}.

The integrator delays in PM4 and the master are respectively $t_{s\,min}$ and $t_{s\,max}$. The last module to finish operation 1 is the master, and it does so exactly t_p later than PM4. Figure 5 shows that PM4 starts operation 2 at a time $t_{s\,min}$ after it completes operation 1, whereas the corresponding figure for the master is $t_{s\,max}$. The argument given earlier showed that if PM4 had not experienced this $t_{s\,min}$ delay, the appropriate value of t_{a3} would have been $t_{s\,max}$. But since PM4 is itself delayed by $t_{s\,min}$, the correct value is:

$$t_{a3} = t_{s\,max} - t_{s\,min} \quad (4)$$

Thus, adding Equations 2, 3, and 4, the correct expression for t_a becomes:

$$t_a \geq 4t_p + t_{s\,max} - t_{s\,min} + (n+1)\,\mathrm{Max}\left[t_d + t_{e,k}\,(0 \leq k \leq n-2), t_f\right] \quad (5)$$

Synchronizing a new module

A Futurebus system has to accept modules being plugged in while the system is "live." To avoid disruption of the bus signals, the module being plugged in, or the newcomer as it is called, must have all its bus-line variables released. Before it can start working normally, the newcomer has to synchronize ap, aq, and ar with the modules that are already working. The 1984 version (p. 40) showed how this could be done, but the method was not fully worked out; it contained several technology dependencies, only one of which was stated explicitly.

Since the 1984 version and Draft 6.2 of the Specification were written, a better scheme has been developed in which the technology dependence is kept to a minimum and is better understood. In fact if two spare bus lines were available, the technology dependence could be avoided altogether. One of the lines would serve as a *request for synchronization* line, which the newcomer would assert after being plugged in. The second line, a *synchronize* line, would be asserted by the current master when it was safe for the newcomer to do so. This would occur during operation 6 of a control acquisition cycle in which the master was relinquishing control of the bus. By asserting this line the master would in effect be telling the newcomer: "The bus is now idle, and I am holding up the acquisition procedure (by not releasing aq) until you have asserted ar and ai." (ai is the inverse address acknowledge signal used in the data-transfer portion of the bus.[10]) When the newcomer had done so, it would indicate the fact by releasing the *request for synchronization* line, and the master would respond by releasing the *synchronize*

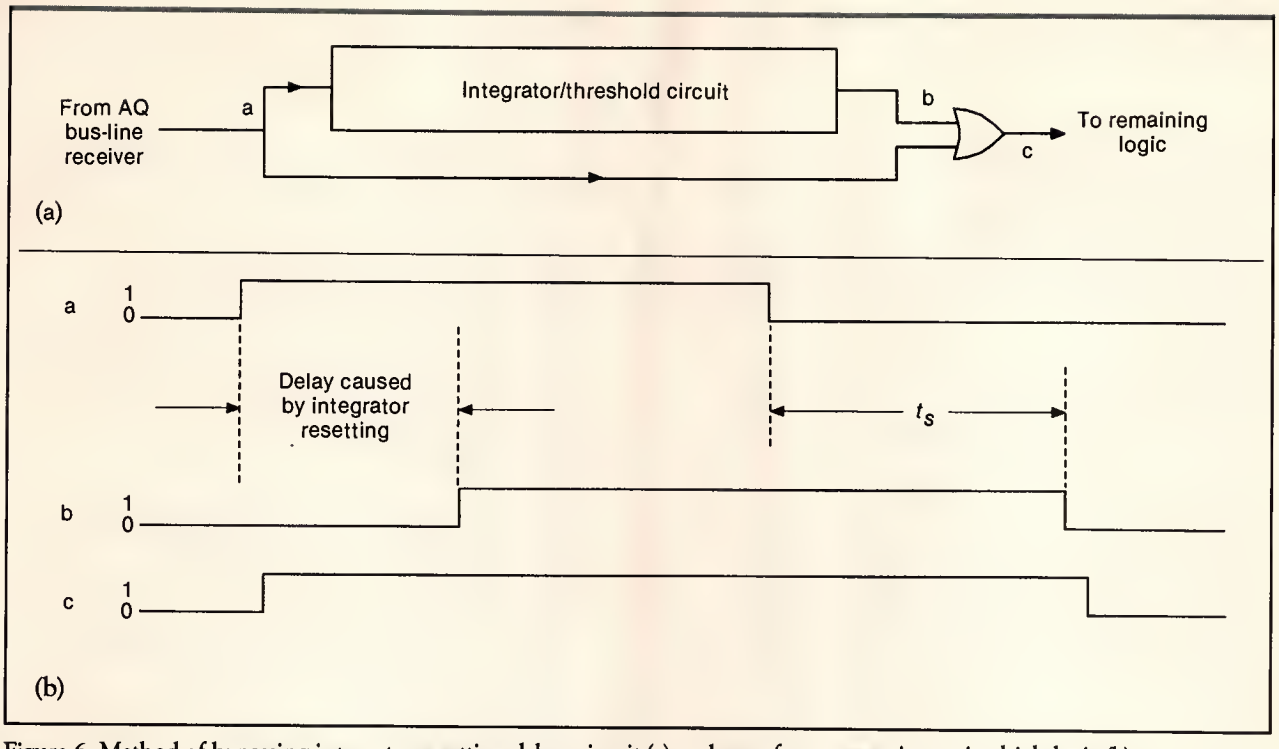


Figure 6. Method of bypassing integrator resetting delay: circuit (a) and waveforms assuming active-high logic (b).

line, after which it would complete operation 6.

Procedure. It would be very attractive to incorporate this scheme into a future version of the specification, but for the time being it is ruled out by a shortage of bus lines. In its place we have used the following procedure, which unfortunately is more complicated and has a small technology dependence.

1) After being plugged in, the newcomer detects the state (AP, AQ, AR) = (0, 0, 1), which indicates either the operation-1-wait state or the operation 3/4 boundary.

2) On the next occasion that it detects AQ asserted, indicating the start of operation 2 or 5, the newcomer asserts its own aq. It needs to have done this within a prescribed time, discussed later, to prevent the rest of the system from proceeding beyond the end of the following operation 3 or 6 as the case may be.

3) After asserting aq, the newcomer waits until it detects the release of AP, which indicates that operation 3/6 has started; it then asserts ar and tests AC. If AC = 0, the operation will be either operation 3 of a fairness-release cycle or an operation 6 in which control of the bus is successfully handed over to a new master. In both cases the next operation is certain to be operation 1.

Therefore if the newcomer finds AC = 0, it joins in the normal control acquisition procedure from the immediately following operation. If on the other hand it finds AC = 1, the following operation could be operation 4 or 1. In this case the newcomer operates the ap, aq, ar protocol but without carrying out any of the normal control acquisition operations. Its only action is to test AC every time (AP, AQ, AR) = (0, 1, 1), that is, in every operation 3 and 6, until it finds AC = 0. When AC = 0, the newcomer joins in the normal control acquisition procedure from the operation 1 immediately following.

At this point the newcomer is not yet ready to take part in data-transfer activity because its ai is still at 0. It asserts ai during the next operation 6 in which mastership is being successfully transferred, that is, an operation 6 in which AC = 0, because as explained above, at that time the data-transfer lines are sure to be idle.

Timing constraints. A study of worst case conditions shows that the maximum time that can be allowed between line AQ at the newcomer switching to 1 and the newcomer's aq being fully switched to 1, that is, including the delays through the newcomer's receiver, some logic, and its transmitter, is:

$$\text{(minimum duration of operation 2 or operation 5)} + 2t_p$$

Minimum duration here means the minimum interval between a module detecting bus line AQ asserted and its releasing ap.

With the bus transceivers presently available, for example, National Semiconductor's DS3896 and DS3897, the sum of the delays through the receiver and transmitter is greater than $2t_p$, which requires the minimum duration of operations 2 and 5 to be greater than zero. The specified value is likely to be about 30 ns, which will allow a maximum total delay through the newcomer's AQ receiver, transmitter, and the intervening logic of 55 ns. There is no need to include the integrator resetting time in this figure if one uses the circuit shown in Figure 6.

This article draws attention to those circumstances wherein the performance of the earlier control acquisition scheme for the IEEE 896 Futurebus could be suboptimum. A modified scheme overcomes the drawback by including the facility for preemption. The advantages that the new plan gives are:

1) a guarantee that, at the end of a module's bus tenure, control of the bus passes to the highest priority module needing it at the time; and
2) emergency-message broadcasting, independently of the normal data-transfer facilities.

A small downward correction has been made in the length of the time interval that modules must allow for the arbitration circuits to settle. Lastly, an improved procedure has been described whereby modules newly plugged into a live system synchronize their operation with the modules already working.

Acknowledgments

The strong advocacy by Keith Britton (Blastmasters Inc.) resulted in the modification of the earlier scheme to include preemption. John Theus (Tektronix Inc.) implemented many features of the new scheme, including preemption. Hugh Field-Richards at the Royal Signals and Radar Establishment,[11] Simon Peyton-Jones at University College, London, Peter Ashenden at the University of Adelaide, and others elsewhere further implemented the new scheme. John Hill (Admiralty Research Establishment) carried out simulation work on the live-insertion arrangements, and the correction to the value of t_a resulted from discussions I had with Michael Rothon (British Telecom).

I should like to record my thanks to all the people mentioned above, to all the other members of the P896 Working Group, and to the director of IBM United Kingdom Laboratories Ltd. for permission to publish this article.

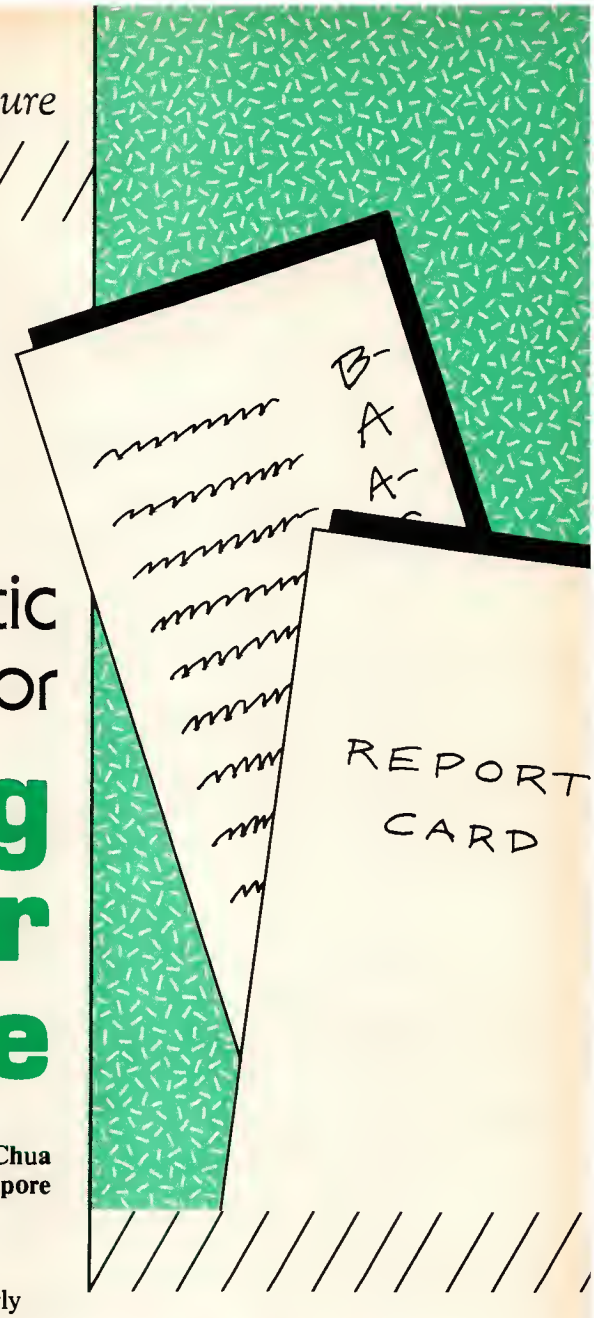
References

1. H. Kirrmann, "Report on the Paris Multibus II Meeting," *IEEE Micro,* Aug. 1985, pp. 82-89.
2. P.L. Borrill, "The 32-bit Bus Standards Issue Revisited," *IEEE Micro,* Oct. 1985, pp. 76-84.
3. P.L. Borrill, "A Comparison of 32-bit Buses," *IEEE Micro,* Dec. 1985, pp. 71-79.
4. P.L. Borrill, "Objective Comparison of 32-bit Buses," *Microprocessors and Microsystems,* Mar. 1986, pp. 94-100. (This is an updated version of Ref. 3).
5. D. del Corso, H. Kirrmann, and J. D. Nicoud, "Microcomputer Buses and Links," Academic Press, New York, 1986.
6. R. Edwards, "Futurebus—the Independent Standard for 32-bit Systems," *Microprocessors and Microsystems,* Mar. 1986, pp. 65-68.
7. S. Peyton-Jones, "Using Futurebus in a Fifth-generation Computer," *Microprocessors and Microsystems,* Mar. 1986, pp. 69-76.
8. D.M. Taub, "Arbitration and Control Acquisition in the Proposed IEEE 896 Futurebus," *IEEE Micro,* 1984, pp. 28-41.
9. I.E. Sutherland et al., "The Trimosbus," *Proc. Caltech Conference on VLSI,* Jan. 1979, pp. 395-427.
10. P. Borrill and J. Theus, "An Advanced Communication Protocol for the Proposed IEEE 896 Futurebus," *IEEE Micro,* 1984, pp. 42-56.
11. H.S. Field-Richards, "A Control-acquisition Interface for Futurebus," RSRE memo. 3935, Royal Signals and Radar Establishment, Malvern WR14 3PS, UK, July 1986.

D. Matthew Taub holds the corporate position of senior technical staff member at IBM United Kingdom Laboratories Ltd. Since 1957 he has worked on magnetic core logic circuits, computer architecture, read-only and magnetic disk storage, peripheral device control using LSI techniques, CRT displays, and design of microcomputer bus systems. He has also worked for Ericsson Telephones Ltd. (now part of the Plessey Group) and Leo Computers Ltd. (now part of ICL).

Taub received the BSc in electrical engineering from University College, Nottingham, and the MSc and PhD from Cambridge University in 1945, 1950, and 1982. He has published 23 articles in journals and 29 articles in the *IBM Technical Disclosure Bulletin* and is named as inventor or coinventor on 28 patents. He is a fellow of the IEE and the British Computer Society and a senior member of the IEEE. From 1977 to 1981, he served as joint honorary editor of *IEE Proceedings on Computers and Digital Techniques.*

Questions concerning this article can be directed to the author at IBM United Kingdom Laboratories Ltd., Mail Point 151, Hursley Park, Winchester, S021 2JN England.



A Synthetic Instruction Mix for Evaluating Microprocessor Performance

John C. McCallum and Tat-Seng Chua
National University of Singapore

Need to rate the performance of that new microprocessor you're interested in? Here's a simple, easy way to do just that.

Estimating the performance of a microprocessor in its design and early production phases can be a relatively easy process as well as a useful one. A manufacturer's documentation, often available to the public before a microcomputer itself is introduced, provides data that can be used in evaluating and comparing the CPU's raw speed.

Here we present a synthetic instruction mix developed for evaluating the performance of microprocessors in scientific, commercial, systems, and general applications. The synthetic instruction mix consists of a set of Move, Add, and Multiply pseudoinstructions based on studies of dynamic statement executions of high-level languages, or HLL. We have translated the pseudoinstructions used in the synthetic instruction mix for several microprocessors to determine the performance in executing the pseudoinstructions for the different types of applications.

Several techniques are known and used for evaluating the performance of computers. Lucas has surveyed the popular evaluation techniques: cycle and add times, instruction mixes, kernel programs, analytic models, synthetic programs, simulation, and performance monitoring.[1]

The major instruction mix formulations were developed by Arbuckle, Knight, and Gibson.[2-4] These early instruction mixes were developed before statistical studies of computer instructions and high-level languages were made by Knuth and others.[5]

Increased interest in determining a mix of instructions that matches HLL use occurred with the arrival of RISC architectures.[6,7] We do not want to com-

0272-1732/87/0600-0063$01.00 © 1987 IEEE

ment on the RISC-versus-CISC controversy.[8] But, our synthetic instruction mix may be able to give measures for comparing the performance of such processors. The examples we give here apply to popular microcomputers.

Any comparative evaluation of different computers can only be definitive in terms of specific programs run under specific conditions. Bell et al. produced a chart comparing several benchmark programs run on a VAX 11/780 and a DECsystem 2060.[9] The chart showed the VAX to be three times slower on one benchmark and 50 percent faster on another. The dependence on unsuitable choices of benchmarks could lead to a factor-of-six difference in expected performance levels. Even within the same computer architecture, different benchmarks can give significantly different results. McCallum's simple test performed on the VAX 8600 and VAX 785 showed a 35-percent difference between performance ratings obtained from the Whetstone and the Sieve benchmarks.[10]

Instruction mixes and benchmarks

The major difference between instruction mixes and benchmark programs for evaluating performance is that the instruction mix looks only at the CPU processor. The benchmark program tests the combination of the language processor and the CPU processor. The language processor causes a larger variation in performance in more cases (see Gilbreath and Gilbreath[11]) than does the CPU performance.

An instruction mix evaluation avoids the language processor problem because its instructions are compiled by hand. A synthetic mix uses pseudoinstructions that can be translated to specific machine language equivalents for different processors.

No method is truly accurate in determining processor performance. However, it is always useful to know the approximate performance of a processor for discussion and comparison purposes. This processor performance measure P is normally derived by running a standard benchmark such as the Whetstone,[12] the Dhrystone,[13] or a more-specialized benchmark suite such as the Unix benchmarks.[14] Despite attempts to claim these benchmark performances as accurate performance ratings, they can provide only a guideline, due to the inclusion of the effect of the language processor.

The Whetstone benchmark derived by Curnow and Wichmann is based on static and dynamic counts of instructions used in the Whetstone Algol system described by Randel and Russell.[12,15,16] The statistics were collected from a total of 949 Algol programs, which were run on the Whetstone system and categorized by counting the number of the Whetstone Algol interpretation instructions. The benchmark was derived from matching a synthetic program to this instruction mix. The Whetstone benchmark, as in any high-level benchmark program, is designed to measure the combined hardware and language processor performance. One significant result of Curnow and Wichmann's work was the measurement of a 7:1 ratio between versions of language processors running on one specific computer (an IBM 360/65). Although the combined language/CPU performance is useful in most evaluation exercises, sometimes only the basic CPU speed is desired—such as when designing a new processor.

The Whetstone benchmark contains three known defects. It lacks a standard mix of operator and operand precisions (word size); its internal loops can be optimized to nothing with good global optimization in the language processor; and its predominance of floating-point operations is too strong.

The difficulty with any benchmark program is that it requires having the processor available to you in the configuration you wish to compare. One often wants to determine the performance of the processor before the processor is actually available. Fortunately, microcomputers in particular often have documentation available before they become accessible to the public. With this data we can create a synthetic instruction mix for determining the performance of microcomputers.

The process of creating a synthetic instruction mix is similar to the problem of creating a synthetic benchmark program. One must determine the function for which the processor will be used to determine what type of instructions will be necessary. Then the frequency of the actual machine instructions must be estimated. The time for executing those instructions in the processor must be determined, given some constraints about the processor's environment. Finally, this process must be repeated for several processors so that the relative performance P is meaningful to people studying the evaluation.

The first stage in developing a synthetic instruction mix is the same as in the case of the synthetic benchmark—to gather statistics on the use of processors. Weicker surveyed data available in the literature for determining an HLL benchmark, Dhrystone (given in Ada and translatable into C and Pascal).[13] The Dhrystone, designed for systems programming, does not include floating-point operations, while the Whetstone is oriented heavily toward such operations. We have surveyed the statistical data on high-level languages for scientific, commercial, and systems applications.

The second stage in developing the synthetic instruction mix is to translate the HLL instructions into corresponding machine language instructions. Compilers translate in the case of synthetic benchmarks. We chose to use a pseudoinstruction form for this "compilation." We derived a set of five pseudoinstructions that must be determined for each pro-

cessor. The five instructions have a variety of data-word lengths to give a total of 12 instructions to estimate for each processor. (Even a small number of instructions can represent the majority of instructions used in a program.[17] We have estimated the weights of these pseudoinstructions for systems programming, commercial programming, scientific programming, and general programming environments. We feel that the general programming environment is the best indicator of performance, in that it will not vary too greatly between one application and another.

The last stage in the benchmark process is to derive the estimated execution time of this mix on specific processors. This stage requires that the pseudoinstructions be translated into actual machine code instructions. We have assumed these pseudoinstructions would have memory-to-memory operations, which means that each pseudoinstruction will likely translate into several microprocessor instructions. Since this is a hand-translation process, we have tried to keep it simple. If the translation were too difficult, few people would be interested in determining performance. For this reason we have kept the number of pseudoinstructions small. The timings of each of these actual machine instructions have to be determined, or estimated where appropriate. In newer microprocessors (such as the Motorola 68020 with instruction caches and pipelined architectures), we must estimate some effective average execution times. When the total times for the execution of the pseudoinstructions have been determined, the times can be put into the instruction mix formula. This gives a performance rating in pMIPS (millions of pseudoinstructions executed per second). This pMIPS rating is about half of the MIPS ratings generally attributed to large computers.[18,19]

Determining the high-level statement mix

Static and dynamic analyses of programs have been performed for many different languages and types of applications. Knuth performed the first major study on language analysis with Fortran.[5] Similar types of studies have been performed for Algol,[15,20] XPL,[21] PL/1,[22] SAL,[23] Cobol,[24,25] Pascal,[26-29] APL,[30] HLL Symbol computer,[31] and Ada.[32,33] Additional work on Fortran has been done by Lurie and Vandoni,[34] Robinson and Torsun,[35] and Partridge and James.[36] Weicker presents a fairly comprehensive summary of HLL instruction statistics on the Dhrystone benchmark.[13]

Some studies of actual machine instructions have been made for a variety of processors. Some of the processors studied were the Maniac and the CDC3600,[17] the IBM S/360,[21] the MOS6502,[37] the VAX,[38] and the Motorola MC68000.[39] Fairclough made static instruction counts on four microprocessors: the TMS9900, the MOS6502, the MC6800, and the MC68000.[7] These studies mainly look at the individual instruction frequencies rather than at the purpose of the instructions. It is therefore difficult to compare instruction frequencies across the processors. Fairclough grouped the instructions into categories that do not correspond directly (but which are easily adjustable) to the HLL statements.

Table 1 summarizes the statement and instruction frequencies found in several empirical studies mentioned above. The majority of these measures are from static frequency counts of high-level languages. However, some machine-level instructions have been counted, and some dynamic instruction counts have been made. These values can be used to calculate

Table 1.
Percentage of statement types in high-level languages.

Form: Language:	Dynamic Fortran[5]	Dynamic Cobol[25]	Dynamic Assembly[39]	Static Fortran[5]	Static Fortran[5]	Static Fortran[35]	Static PL/1[21]
Statement							
Assign	67			51	41	38	41
Add		20	16				
Move		27	33				
If	11	33	11	10	15	9	18
Goto	9	11	16	9	13	9	12
Do	3		8	9	4	6	7
Call	3		6	5	8	3	2
Perform		6					
Total%	93	97	90	84	81	65	80
No. lines	15,000 est*	21,745	2,000 est	15,000 est	250,000	29,971	145,994

Table 1.
Percentage of statement types in high-level languages. (cont'd.)

Form: Language:	Static Cobol[24]	Static Cobol[25]	Static Assembly[7]	Static Pascal[29]	Static Pascal[40]	Static Pascal[27]	Static Pascal[28]
Statement							
Assign				42	49	34	44
Add	7	8	14				
Move	38	34	45				
If	15	18	10	14	9	18	14.8
Goto	14	17	14	0.5	0.3	0.3	0.3
Do			6	8.1	11	7	8.2
Call			9	34.3	9	40	17.5
Perform	11	13					
Total%	85	90	98	98.9	78.3	99.3	84.8
No. lines	226,466	21,745	50,000 est	11,393 est	2,000	24,512	59,018

*Where the number of lines of code was unknown, we have estimated.

several HLL statement mixes. We have used the five types of high-level statements in our mix: Assignment, If, Goto, Do, and Call. See Table 2 for a list of the calculated statement mixes. We based our choice of limiting the statement types on Knuth's Fortran data in which 75 percent were all Fortran statements and another 20 percent were nonexecutable statements.[5] We omitted only input/output statements from the major instructions. I/O instructions were thought to be too variable among machines to determine actual machine translations. This choice of statement types led to a problem in some languages such as Cobol and assembly in which the Move and Compute categories of instructions would correspond to assignment and similar equivalence problems. We tried to translate these reasonably.

We calculated the percentages of dynamic statement frequency separately from the static statement percentages. The quoted statement types and frequencies in Table 1 have been adjusted to fit into our categories. The total statement frequencies do not equal 100 percent for different reasons: nonexecutable statements and the limited statement set we have chosen. As a result we adjusted the statement frequencies to give totals of 100 percent.

We then weighted the normalized statement frequencies by the sample size of the studies and made total counts of the weighted instructions. Finally, we adjusted the Add, Move, and Perform statements to fit into the Assign, Do, and Call categories.

Table 2 summarizes the percentage of statement types found in static and dynamic execution counts. The static statistics are calculated from a sample size of about 850,000 lines of code. The dynamic results are generated by about 40,000 lines of code.

Table 2.
Normalized statement frequencies (in percentages) for a general mix.

Statement	Static	Dynamic
Assignment	52	58
If	18	24
Goto	13	11
Call	10	4
Do	7	3
Total%	100	100

Determining operations in language statements

We have estimated the major statement frequencies for Assignment, If, Goto, Do, and Call statements. These statements must now be studied to determine the pseudoinstructions that they each generate. The generated instructions depend on the type of operators and operands being used in the statements. Weicker's survey includes operand types for some high-level languages. It is difficult to translate the HLL operands directly to low-level operand types. However, we have tried to estimate this split from the gathered statistics.

We estimated how many 64-bit, 32-bit, 16-bit, and 8-bit operands exist so we could determine typical Move operations.[13] (We included 104-bit, or 13-byte, operands due to the average Cobol Move operand found by Torsun and Al-Jarrah.[25]) The split of operands among integer, real and double-precision, and complex values indicates the complexity of the floating-point operations that might have to be performed. The major study of operand types is an unpublished 1980 study by Patterson and quoted by Weicker.[13]

Lurie and Vandoni studied scientific Fortran identifier names based on 92,463 lines of code in CERN's program library.[34] In Fortran, the implicit name convention for integers is almost always followed by programmers. Lurie and Vandoni found that approximately 40 percent of the occurring identifiers start with the letters I to N. These identifiers did not include the Fortran keywords and function names, which they counted separately. The relative occurrences of the double-precision function names and single-precision function names of sin, abs, and cos (6.7 percent, 4.5 percent, and 7.2 percent) give an estimate of double precision to real usage. The exp and abs functions give a similar value for the complex to real usage (5.2 percent and 7.1 percent). This results in the split of real (53 percent), integer (40 percent), double (3.5 percent), and complex (3.5 percent) for heavy scientific computing from a static Fortran program analysis. We have included the complex type into the double-precision category to simplify the analysis.

The percentages of operand types used in systems programs are not so easy to estimate. The reason for this difficulty is the heavy use of string or character variables. Even assembly language in CISC machines such as the VAX can give problems in evaluation due to the multibyte Move operations. It appears we have no choice other than to crudely estimate a split among the operand types, taking into account Weicker's quotations and looking at VAX opcode distributions.[38] Table 3 lists the distribution of elementary operand types we have used.

Having determined estimates for the mix of high-level statements and the percentages of operands used in the statements, we must determine the mix of operators. We used only the Assignment statement or its equivalent to determine the operator frequencies.

Determining operator frequencies

The translation of the Assignment statement to pseudoinstructions requires the knowledge of the frequency of Assignment statements in programs (given in Table 2); the relative distribution of the types of operands (given in Table 3); the relative distribution of the types of operators; and the relative distribution of the forms in which the Assignment statement appears. We determine the distributions of the types of operators and the forms of the Assignment statement in this section. These distributions differ among programs written for scientific, commercial, and systems programming applications, so we consider each separately.

Table 3.
Normalized distribution of operand types by application.

	Bits	Scientific[1]	Commercial[2]	Systems[3]	General[4]
Quadword					
Double-precision	64	7	0	0	2
Group(str,rec)	104		59	4	21
Long word					
Real	32	53	0	0	18
Integer/address/ pointer	32		9	5	5
Word					
Integer	16	40	9	54	34
Byte made up of:		0	23	37	20
Character	8		(22)	(20)	
Enumeration	8		(1)	(12)	
Boolean	8			(5)	
Total %		100	100	100	100

[1] Derived from variable and function name distributions (see text).
[2] Estimated from Torsun and Al-Jarrah.[25]
[3] Modified from the Dhrystone benchmark.
[4] An average of the other three distributions.

Table 4.
Static frequencies (in percentages) of operators in Assignment statements or equivalents. (Derived from Knuth;[5] functions not included.)

		Scientific	Commercial	Systems	General
Operator	Number	Relative frequency	Relative frequency	Relative frequency	Relative frequency
+	10,593	23	19	35	26
+1	7,200	15	74	24	38
—	10,298	22	3	26	17
*	12,348	27	3	11	14
/	4,739	10	1	4	5
**	681	2			1
**2	427	1			
Total %		100	100	100	100

Scientific programs. Several studies have counted operator frequencies in statements (see Weicker for a summary[13]). Knuth has made static counts of the operators in Assignment statements in scientific programs.[5] Knuth found that 68 percent of all assignments were of the form A = B, and that 12.5 percent were of the type A = A op B. Using his complexity values for the Assignment statements, we can estimate that 11 percent were of the form A = B op C, in addition to the previous 12.5 percent. Knuth also found that 40 percent of additions were simply incremented by one. Interpretation of his table shows that 3.9 percent of statements fall into the form A = B op C op D. Only 4.6 percent of statements have more operations, and we assumed that there were four operands to simplify further estimations. The relative percentages of operators that Knuth found are given in Table 4.

Commercial programs. For comparison of statement types, we combined the Cobol Move and Add statements and said they were equivalent to an Assignment statement. Here we have to consider the original statement types. Torsun and Al-Jarrah did a thorough dynamic analysis of Cobol programs.[25] They found that in the Move statement (26.9 percent), the average number of operands was two, or basically, one string is moved to another location, where the string is about 13 bytes long. A small percentage of these moves implied a translation of types. In the Add statement (19.8 percent), about 80 percent of the executed instructions were similar to an increment (add a constant to a computational variable). We attribute the remaining 20 percent of Add statements to the form C = A + B. We consider the Subtract statement (0.7 percent) and the Multiply statement (0.6 percent) to be two-operand statements as well. Compute (0.4 percent) will be considered to be a three-operand statement.

Systems. We have taken the arithmetic operator frequencies from the Dhrystone benchmark (note that Table XI of Weicker's paper contains some typographical errors). We have assumed that 40 percent of the additions are increments to correspond with the Fortran findings. Table 4 lists the operator frequencies. The distribution of forms of the Assignment statement is also taken from the Dhrystone benchmark (adapted from Weicker's Table VIII). Table 5 summarizes the forms of the Assignment statements that we have used in the synthetic instruction mix.

Pseudoinstructions

We have determined the operations that are to be performed and the operand types on which the operations are being performed. Now, we can determine what pseudo-operations should be used in the synthetic instruction mix. Table 6 summarizes the pseudoinstructions. These instructions are all memory-to-memory instructions, since the majority of HLL operations work directly with memory.

In choosing these pseudoinstructions, we took into account many reasons. Consider each of the pseudoinstructions separately.

- The Move8 instruction implements the character, enumeration, and Boolean operand moves. It is also used for smaller portions of the string movement. In 8-bit processors, the 8-bit Move gives a significant advantage to comparisons using 16-bit Moves.
- Move16 is the major Move instruction for integer numbers. Most integers can be represented with a

Table 5.
Forms of the Assignment statement or equivalent. Frequencies are stated in percentages.

Form	Scientific[1]	Commercial[2]	Systems[3]	General[4]
A = B	68	55	66	64
A = A op B	12.5	33	6	17
A = B op C	11	11	24	15
A = B op C op D	3.9	1	2	2
A = B op C op D op E	4.6	0	2	2
Total %	100	100	100	100

[1] Based mainly on Knuth's Fortran data.[5]
[2] Based on data from Torsun and Al-Jarrah.[25]
[3] Based on the Dhrystone systems benchmark.
[4] The average of the other three types. The probabilities of these forms is adjusted in the overall mix by the probability of the Assignment statement.

Table 6.
Pseudoinstructions used in the synthetic instruction mix.

	Form
Move8	A to B
Move16	A to B
Move32	A to B
Load32	A to Reg
Incr16	A + 1 to A
Add8	A to B
Add16	A + B to C
Add32	A + B to C
Fadd32	A + B to C
Fadd64	A + B to C
Mul16	A * B to C
Fmul32	A * B to C

16-bit value. Turbo-Pascal for example, does not allow a 32-bit integer representation.[41] Move16 gives a 16-bit processor a significant advantage in comparison with 32-bit processors.

• The Move32 instruction moves a variety of operands: long integer and standard reals, as well as addresses during a Call. Long string Moves are made up of three of these Moves. The "integers" represented by the 32-bit values come from the commercial area where more accurate values are required for fixed decimal representation. Repetitions of this 32-bit Move make up long string Moves. (If 64-bit processors become available, a new synthetic benchmark should be formulated with a 64-bit Move.)

• Load32 moves an address into a register. This instruction simulates a Goto instruction, which involves loading a register. It is also used in the Call statement equivalent. We use a 32-bit value based on large memory operation or an equivalent.

• The Incr16 is the only one-address instruction; all other instructions involve two or three addresses. Loops are normally controlled by integer indexes. The arithmetically frequent increment translates into a much faster instruction than an Add in most cases. Since integers are normally 16-bit values, we did not choose a 32-bit increment.

• The Add8 instruction should really be a Subtract, used to compare two values. But we assume that an Add and a Subtract instruction are roughly equal in timing; only Add (and not Subtract) has been included in the various word sizes and forms.

• Add16 is the major integer computational instruction. Like all of the following instructions, it is a three-operand memory-to-memory operation.

• Add32 is used mainly for commercial programming where larger integers keep accuracy for accounting purposes. This instruction is also used in place of the Subtract instruction for testing equality in If statements. Since addresses are assumed to be 32-bit values, any address calculations, such as those using strings, include 32-bit operations.

• The Fadd32 floating-point addition is used almost exclusively in the scientific area. However, it usually requires significantly different computing time than does 32-bit integer Adds and so must be used in benchmarking for scientific and general benchmarking.

• The Fadd64 double-precision floating-point operations are not common, but they do take considerably more time than regular floating-point operations. The double-precision floating-point

Multiply and Divide instructions have been translated to Fadd64, since the evaluation times should be similar and the relative frequencies of the other double-precision operations is small.

- In Mul16 two 16-bit numbers are multiplied to produce a 16-bit result. The execution time of this instruction is likely to be similar to a 16-bit Divide, so 16-bit divisions also are translated into the Mul16 pseudo-operation. We mapped all operand sizes of integer multiplication and division to this instruction.
- In Fmul32 two real numbers are multiplied to produce a resulting 32-bit number. Fmul32 has been used to represent real 32-bit division as well.

Usually, the instructions do not translate according to a strict set of rules when considering specific microprocessors. This happens because the processors are usually a mix of 8-, 16-, and 32-bit instructions.

Table 7.
Pseudoinstructions due to A = B or Move command.

	Operand length				
Bits	8	16	32	64	104
Move8	1	0	0	0	1
Move16	0	1	0	0	0
Move32	0	0	1	2	3

Determining the synthetic mix of pseudoinstructions

Using the statement frequencies from Table 2 and from Table 5, we can generate a set of the frequencies of the statements of different forms for the four application categories. The problem then is to translate these forms of statements into our pseudoinstructions. Each of the statement forms will generate a different set of pseudoinstructions, so consider each statement form separately.

- The A = B statement form is simply a Move instruction. In our pseudoinstructions, it generates one or more Move commands of a length depending on the lengths of the operand. The number of Move instructions generated for the type of operand is given in Table 7.
- A = A op B is a fairly standard two-operation instruction between two operands stored in memory. Although the two address forms of actual machine instructions are significantly different from the three-address forms, we decided to keep the instruction mix reasonably simple. We assumed that this two-operand form is equivalent to the time for executing a three-operand form, minus one half of the time to execute a Move instruction of the proper length.
- A = B op C is the standard memory-to-memory, three-address instruction form. Typically in a two-address machine it compiles to a sequence of the form:

  Load B to register;
  Operate from C into register;
  Store register to memory location A.

Table 8.
Translations of operator/operand pairs into pseudoinstructions.

	Operand type					
Operator Form	Byte	Int16	Int32	Real32	Real64	Str104
+	Add8	Add16	Add32	Fadd32	Fadd64	3*Move32 + Move8
+1	Add8	Incr16	Add32	Fadd32	Fadd64	Add32
—	Add8	Add16	Add32	Fadd32	Fadd64	3*Add32 + Add8
*	Mul16	Mul16	Mul16	Fmul32	Fadd64	Add32
/	Mul16	Mul16	Mul16	Fmul32	Fadd64	Add32
**	2*Move8	5*Fadd32 +6*Fmul32 —5*Move32	5*Fadd32 +6*Fmul32 —5*Move32	5*Fadd32 +6*Fmul32 —5*Move32	17*Fadd64 —18*Move32	Add32
**2	Mul16	Mul16	Mul16	Fmul32	Fadd64	Add32

Table 9.
Number of pseudoinstructions generated by the translation of the If statement.

Pseudo-instruction	Operand type					
	Byte	Int16	Int32	Real32	Real64	Str104
Load32	1	1	1	1	1	1
Move32	0	0	0	0	−0.5	−1
Add8	1	0	0	0	0	1
Add16	0	1	0	0	0	0
Add32	0	0	1	1	2	3

The instructions that this form of statement generates are shown in Table 8.

- A = B op C op D is similar to the standard arithmetic Assignment statement shown above. However, it has an additional operation that should be made from memory to register. We estimated our instructions for memory-to-memory operation. Our equivalent is obtained by performing two standard A + B op C operations, minus one half of a Move operation. Even on a processor with a three-address architecture, this should give a fair representation of the instruction timing for the statement. We assume that the operand type is the same for all operations in the statement.
- For A = B op C op D op E we use the same assumptions as above and assume three times the instruction weighting from Table 8 and subtract one Move of the corresponding operand length. These multiple-operation statements occur infrequently, ensuring that few errors should be introduced by this approximation.
- In the If statement we assume that a comparison and a choice of program flow is made. This corresponds roughly to a subtraction and a register (program counter) load. The comparison must be done for the length of the operand. However, floating-point operations are not necessary even for non-equality testing. Table 9 shows the translation of the If statement into pseudo-operations.
- The Goto statement is simply a Load32 instruction, since its function is to transfer data to a specific location in full memory.
- We assume that the Call statement translates into two Load32 instructions and two Move32 instructions to perform two transfers and two parameter stores. Although it is likely we will find a stack instruction, this should give us a reasonable equivalent. Using a machine Call instruction is not likely to give a better result because of the variety of stack operations or transfers occuring in high-level programming languages.
- The Do or loop instruction increments a counter, comparing the counter with some control value and branching to some location in memory. Therefore we translated it as an Incr16, a Compare in the form of Add16, and a branch in the Load32 form. These translations are independent of operand statistics, since loops are most frequently integer loops.

Generating the pseudo-operation mix

Now that we have stated the basic translations, we must perform the calculations for each of the application areas so they can be translated into the pseudoinstruction mix. We do this by calculating the number of occurrences of each of the pseudo-instructions from the previous tables and writing descriptions of the instructions. Table 10 lists the formulae used in these calculations.

The formulae in Table 10 are based on the probabilities of: the statement forms such as p(A = B) from Tables 2 and 5; the operand types such as p(byte) from Table 3; and the operator types such as p(+1) from Table 4. We developed the equations from the explanation in this section and Tables 7, 8, and 9, which explain how the statement types get translated to the pseudoinstructions.

When we use the dynamic instruction frequencies from Table 2 in the preceding equations along with the probabilities in Tables 3, 4, and 5, we achieve the relative instruction frequencies. We normalize these frequencies, so that we obtain the probability distributions of the different instruction types (of our

Table 10.
Pseudoinstruction counts based on probabilities of operators, operands, statement frequencies, and forms.

```
MOVE8 = p(A=B)*{p(byte)+p(str104)}
      - {p(byte)*[p(+)+p(+1)+p(-)] + p(str104)*p(+)}
         * {p(A=AopB)*.5+p(A=BopCopD)*.5+p(A=BopCopDopE)}
      + {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
         * {2*p(byte)*p(**) + p(str104)*p(+)}

MOVE16 = p(A=B)*p(int16)
       -{p(int16)+[p(byte)+p(int32)]*[p(*)+p(/)+p(**2)]}
          *{p(A=AopB)*.5+p(A=BopCopD)*.5+p(A=BopCopDopE)}

MOVE32 = p(A=B)*{p(int32)+p(real32)+2*p(real64)+3*p(str104)}
   -{p(int32)+p(real32)+2*p(real64)}
     *{p(A=AopB)*.5+p(A=BopCopD)*.5+p(A=BopCopDopE)}
   + {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     *{p(str104)*3*p(+)
        - p(**)*[5*[p(real32)+p(int32)+p(int16)]+18*p(real64)]}
   - p(IF)*{p(real64)*.5 +p(str104)}
   + 2*p(CALL)

LOAD32 = p(IF) + p(GOTO) + 2*p(CALL) + p(DO)

INCR16 = p(int16)*p(+1)
     *{p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     + p(DO)

ADD8 = {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
          * {p(byte)*[p(+)+p(+1)+p(-)]+p(str104)*p(-)}
          + p(IF)*p(str104)

ADD16 = {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     *p(int16)*{p(+)+p(-)}
     + p(IF)*p(int16) + p(DO)

ADD32 = {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     * {p(int32)*[p(+)+p(+1)+p(-)]
        +p(str104)*[p(+1)+3*p(-)+p(*)+p(/)+p(**)+p(**2)]}
     + p(IF)*{p(int32)+p(real32)+2*p(real64)+3*p(str104)}

FADD32 = {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     * {p(real32)*[p(+)+p(+1)+p(-)]
          +5*p(**)*[p(real32)+p(int32)+p(int16)]}

FADD64 = {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     * p(real64)*{p(+)+p(+1)+p(-)+p(*)+p(/)+p(**2)+17*p(**)}

MUL16 = {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     * [p(*)+p(/)+p(**2)] * {p(byte)+p(int16)+p(int32)}

FMUL32 = {p(A=AopB)+p(A=BopC)+2*p(A=BopCopD)+3*p(A=BopCopDopE)}
     * {p(real32)*[p(*)+p(/)+p(**2)]
         + 6*p(**)*[p(real32)+p(int32)+p(int16)]}
```

pseudoinstructions) in each type of computing. These are given in Table 11.

Table 11 does not show the relative frequencies of executing these instruction types in these applications. Instead, the table lists the weightings of this set of instructions to determine the relative performance of the processors as a whole. The assumptions made in subtracting Move instructions to remove the effect of the storing of intermediate values in Assignment statement evaluation means that the distribution is not correct for Moves. Moves have been traded off for the more-powerful addressing modes of three-operand instructions.

Considering this limitation of the synthetic instruction mix, we can compare our mix with the other instruction mixes that have been used in performance evaluation. The other values in Table 12 have been taken from Bell et al.[9]

It is difficult to compare the instruction mixes because (1) many instruction types may be equivalent for execution, and (2) seemingly similar instruction types may require widely varying execution times.

Table 11.
Distribution of pseudo-operations used in mixes.

Pseudo-operations	Scientific	Commercial	Systems	General
Move8	0	13.7	11.6	9.6
Move16	11.5	1.6	15.4	8.1
Move32	21.7	26.5	9.9	21.8
Load32	28.1	19.4	30.9	24.4
Incr16	2.8	2.0	4.1	3.3
Add8	0	8.7	5.8	5.2
Add16	10.6	2.4	15.8	7.8
Add32	9.8	25.5	4.2	14.9
Fadd32	6.6	0	0	2.3
Fadd64	1.5	0	0	0.3
Mul16	2.4	0.2	2.3	1.4
Fmul32	5.0	0	0	0.9
Total%	100	100	100	100

Table 12.
Instruction mix weights.

Instruction	Arbuckle[2]	Gibson[4]	Knight[3] Scientific	Knight[3] Commercial	This work General
Fixed +/—		6	10(25)	25(45)	31
Multiply		3	6	1	1.4
Divide		1	2		
Floating +/—	9.5		10		2.6
Floating multiply	5.6				0.9
Floating divide	2.0				
Load/store/move	28.5	25			64
Indexing	22.5				
Conditional branch	13.2	20			
Compare		24			
Branch on character		10			
Edit		4			
I/O initiate		7			
Other	18.7		72	74	

Table 13.
Clock cycles required to execute the pseudoinstructions on various microprocessors (and the MV/8000).

Pseudo-instruction	I8085	I286/287*	MC68000	MC68020	MV/8000
Move8	26	16	20	9.5	1.1
Move16	32	16	20	9.5	0.88
Move32	64	32	28	9.5	1.32
Load32	36	10	16	6	0.66
Incr16	38	14	16	9	1.32
Add8	43	30	36	17	1.76
Add16	62	30	36	17	1.54
Add32	139	60	50	17	6.82
Fadd32	2000	650	232	150	2.64
Fadd64	4000	780	464	300	43.34
Mul16	1400	64	102	42	3.41
Fmul32	2500	688	894	300	3.96

*The I286/287 time is quoted in crystal clocks instead of system clocks. These times must be multiplied by 1.05 to get average times due to instruction fetches.

Table 14.
The pMIPS ratings determined for various microprocessors.

Processor	I8085	I286/287	MC68000	MC68020	MV/8000
Speed (MHz)	5	16*	8	16.67	1
Pseudo-instruction					
Scientific	0.01	0.14	0.09	0.43	0.40
Commercial	0.07	0.48	0.26	1.44	0.38
Systems	0.06	0.70	0.31	1.50	0.71
General	0.03	0.31	0.19	0.93	0.45

*Xtal frequency ratings are given for processors using memory with no wait states.

Translating to specific microprocessor instructions

Now that we have determined the relative instruction mix, we must look at the time it takes to execute these instructions on various microprocessors. Here, we look specifically at the Intel 8085,[42] the Intel 80286/80287,[43] the Motorola 68000,[44] and the Motorola 68020.[45] In addition, we have included the Data General MV/8000 as a comparison with a standard super-minicomputer.[46] The pseudoinstruction translations to the actual machine code appear in the accompanying box. Table 13 lists the number of clock cycles required to execute each pseudoinstruction on each of the processors. The MV/8000 data appears as actual average execution times in microseconds, since its clock is not adjustable by users.

When we include these execution time values in the instruction mix weightings, we obtain the performance values given in Table 14. The performance values are quoted in pMIPS, which are the millions of the pseudoinstructions executed per second. These pseudoinstructions are carried out from memory to memory, actions which require more time to execute than the simpler register-to-register operations. High-level languages do most of their work from memory to memory, as shown previously. These pMIPS ratings reflect performances of more than the instructions executed in most computers. They can be expected to be slower to execute than register-to-regis-

Translation of pseudoinstructions into specific assembly language instructions for selected processors

The time to execute each of the 12 pseudoinstructions must be determined for each microprocessor of interest. Each pseudoinstruction should be translated into the fastest set of assembly language instructions for the specific microprocessor. The clock cycle time required to execute each of the assembly language instructions is usually found in the manufacturer's documentation. Some estimation may be necessary for difficult instructions or to adjust for the effects of cache memory and pipelined instruction execution.

Here we give the assembly language instructions and the execution time for the instructions in system clock cycles (based on no wait states for memory accesses) for several microprocessors: the Intel 8085, the Intel 80286/80287, and the Motorola 68000 and 68020. The MV/8000 super-minicomputer execution times are also given for comparison. The timing results for the pseudoinstructions are summarized in Table 13.

The Intel 8085, an updated version of the 8080 microprocessor, is also similar to the Z80 microprocessor. All three have 8-bit processors with some limited 16-bit processing capabilities. The standard fast version of the 8085 operates at 5 MHz. A version of the Z80 processor operates at 10 MHz. Please refer to Table A.

In the calculations for the Intel 80286/80287 shown in Table B, we use the basic system clock, since it is divided by two for the CPU and by three for the floating-point processor. This is the same as calling the IBM PC AT a 12-MHz processor. With this system clock, there are four clocks to the bus (memory) cycle. We use real address mode timings instead of virtual address mode timings. The instruction clock timings are ideal timings. Intel suggests that 5 percent be added for instructions that execute faster than they can be fetched. The timings for the individual instructions are given, then the adjustment is added into the total weighted instruction execution time.

Table A.
The Intel 8085.

Pseudo-instruction		Machine code		Clock cycles	
Move8		LDA data1		13	
		STA data2		13	
Move16		LHLD data1		16	
		SHLD data 2		16	
Move32		LHLD data1		16	
		SHLD data2		16	
		LHLD data1 + 2		16	
		SHLD data2 + 2		16	
Incr16		LHLD data1		16	
		INX H		6	
		SHLD data1		16	
Load32		LHLD data1		16	
		XCHG		4	
		LHLD data1 + 2		16	
Add8		LDA data1		13	
		LXI #data2,HL		10	
		ADD M		7	
		STA data3		13	
Add16		LHLD data1		16	
		XCHG		4	
		LHLD data2		16	
		DAD D		10	
		SHLD data3		16	
Add32		LHLD data1		16	
		XCHG		4	
		LHLD data2		16	
		DAD D		10	
		SHLD data3		16	
		LHLD data1 + 2		16	
		XCHG		4	
		LHLD data2 + 2		16	
		JNC next	(7 + 10) / 2 =	9	
		INX H		6	
	next	DAD D		10	
		SHLD data3 + 2		16	139
Fadd32		est			2000
Fadd64		est			4000
Mul16		est			1400
Fmul32		est			2500

Table B.
The Intel 80286/80287.

Psuedo-instruction	Machine code	pclocks	System clocks	
Move8	MOV data1, AL	5	10	
	MOV AL,data2	3	6	
Move16	MOV data1,AX	5		
	MOV AX,data2	3	16	
Move32	MOVE16			
	MOVE16	16	32	
Load32	MOVE data,AX	5	10	
Incr16	INC data	7	14	
Add8	MOV data1,AL	5		
	ADD data2,AL	7		
	MOV AL,data3	3	30	
Add16	MOV data1, AX	5		
	ADD data2,AX	7		
	MOV AX,data3	3	30	
Add32	MOV data1,AX	5		
	ADD data2,AX	7		
	MOV AX,data3	3		
	MOV data4,AX	5		
	ADC data5,AX	7		
	MOV AX,data5	3	60	
Fadd32	FLD data1,ST(0)	38-56	141	
	FADD data2,ST(0)	90-120	315	
	FST ST(0),data3	84-90	194	650
Fadd64	FLD data1,ST(0)	40-60	150	
	FADD data2,ST(0)	95-125	330	
	FST ST(0),data3	96-104	300	780
Mul16	MOV data1,AX	5		
	IMUL data2	24		
	MOV AX,data3	3	64	
Fmul32	FLD data1,ST(0)	38-56	141	
	FMUL data2,ST(0)	110-125	353	
	FST ST(0),data3	84-90	194	688

Table C.
The Motorola 68000 and 68020.

Psuedo-instruction	Machine code		68020 Best	68020 Worst	68020 Cache	68000
Move8	MOVE.B	A0@(data1),A0@(data2)	6	13	8	20
Move16	MOVE.W	A0@(data1),A0@(data2)	6	13	8	20
Move32	MOVE.L	A0@(data1),A0@(data2)	6	13	8	28
Load32	MOVE.L	A0@(data1),A1	3	9	7	16
Incr16	ADDQ.W	# <1>,A0(data1)	6	12	9	16
Add8	MOVE.B	A0@(data1),D1	3	9	7	12
	ADD.B	A0@(data2),D1	3	9	7	12
	MOVE.B	D1,A0@(data3)	3	7	5	12
Add16	MOVE.W	A0@(data1),D1	3	9	7	12
	ADD.W	A0@(data2),D1	3	9	7	12
	MOVE.W	D1,A0@(data3)	3	7	5	12
Add32	MOVE.L	A0@(data1),D1	3	9	7	16
	ADD.L	A0@(data2),D1	3	9	7	18
	MOVE.L	D1,A0@(data3)	3	7	5	16
Fadd32	est from Motorola software			150		232
Fadd64	est			300		464
Mul16	MOVE.W	A0@(data1),D1	3	9	7	12
	MUL.W	A0@(data2),D1	28	34	32	78
	MOVE.W	D1,A0@(data3)	3	7	5	12
Fmul32	est			300		894

Table C presents calculations for the Motorola microprocessors. Internally, the Motorola 68000 is a 32-bit processor; externally it is a 16-bit processor. The 68020, however, is totally a 32-bit processor. The two machines are compatible in software.

The Motorola 68020 has an instruction cache that allows instructions to execute faster than could be done from memory. The specifications quote three timings for the 68020's instruction execution clock cycles: best, worst, and cache. We quote all the times on the chart, but we average the best and worst case to estimate performance.

The Data General MV/8000, an older, 1-MIPS super-minicomputer, is a two-address, 32-bit extension to the Nova and Eclipse architectures. We quote the timings shown in Table D in micro-seconds for average instruction execution time.

Table D.
The Data General MV/8000.

Psuedo-instruction	Machine code	Timings
Move8	LLDB data1,A	0.44
	LSTB A,data2	0.66
Move16	LNLDA data1,A	0.44
	LNSTA A,data2	0.44
Move32	LWLDA data1,A	0.66
	LWSTA A,data2	0.66
Load32	LWLDA data1,A	0.66
Incr16	LNISZ data1	1.32
Add8	LLDB data1,A	0.44
	LNADD data2,A	0.66
	LSTB A,data3	0.66
Add16	LNLDA data1,A	0.44
	LNADD data2,A	0.66
	LNSTA A,data3	0.44
Add32	LWLDA data1,A	0.66
	LWADD data2,A	5.50
	LWSTA A,data3	0.66
Fadd32	LFLDS data1,FPAC	0.66
	LFAMS data2,FPAC	1.54
	LFSTS FPAC,data3	0.44
Fadd64	LFLDD data1,FPAC	1.10
	LFAMD data2,FPAC	41.36
	LFSTD FPAC,data3	0.88
Mul16	LNLDA data1,A	0.44
	LNMUL data2,A	2.53
	LNSTA A,data3	0.44
Fmul32	LFLDS data1,FPAC	0.66
	LFMMS data2,FPAC	2.86
	LFSTS FPAC,data3	0.44

ter, or memory-to-register operations. This fact illustrates the vagueness of the definition of MIPS.

When trying to quote a single value for the performance of a processor, it is best to choose the general-performance pMIPS rating. Since the MV/8000 is generally regarded as being a 1-MIPS processor, the pMIPS rating is about half of a generally quoted MIPS rating. The general pMIPS rating is based on executing the more powerful memory-to-memory pseudoinstructions, rather than the faster, but less-powerful, register-based instructions.

We have developed a method for estimating the performance of a microprocessor using a synthetic instruction mix. The synthetic instruction mix consists of a set of Move, Add, and Multiply instructions. We chose pseudoinstructions based on high-level-language considerations and assumptions for two reasons. We tried to simplify the set and ensure that the resulting pseudoinstructions would be simple to determine for specific processors. A major decision consideration in the determination of the pseudoinstructions was to include various operand lengths in the instruction set.

A review of the statement forms used in high-level languages was used to estimate statement distributions. We determined operator and operand distributions from studies of scientific, commercial, and systems programs. From these statistics we could determine equations for converting these application types to relative instruction mixes of the pseudoinstructions.

We used specific processors as examples in trying to determine the performance, or pMIPS, ratings. Next, we converted pseudoinstructions to specific processor assembly language instructions and determined the clock cycles required to execute each pseudoinstruction. We then applied the synthetic instruction mix to the pseudoinstruction execution times and obtained the pMIPS rating for the different application areas.

The major advantages of the proposed synthetic instruction mix are:

- It is relatively easy to evaluate;
- It determines the raw speed of the processor rather than evaluating the combined CPU and language processor pair; and,
- It can be used to evaluate a processor that is not available physically, such as during design and early production phases of a microprocessor.

Many weaknesses can be found in the methods used to determine these pMIPS ratings. However, any method of performance evaluation is open to criticism. We feel that the pMIPS rating gives a good comparative rating among processors.

References

1. H.C. Lucas, Jr., "Performance Evaluation and Monitoring," *Computing Surveys,* Sept. 1971, pp. 79-91.
2. R.A. Arbuckle, "Computer Analysis and Thruput Evaluation," *Computers and Automation,* Jan. 1966, pp. 12-19.
3. K.E. Knight, "Changes in Computer Performance," *Datamation,* Sept. 1966, pp. 40-54.
4. J.C. Gibson, "The Gibson Mix," IBM tech. report TR2043, June 18, 1970.
5. D.E. Knuth, "An Empirical Study of Fortran Programs," *Software P & E,* 1971, pp. 105-133.
6. D.A. Patterson and C.H. Sequin, "A VLSI RISC," *Computer,* Sept. 1982, pp. 8-21.
7. D.A. Fairclough, "A Unique Microprocessor Instruction Set," *IEEE Micro,* May 1982, pp. 8-18.
8. R.P. Colwell et al., "Computers, Complexity and Controversy," *Computer,* Sept. 1985, pp. 8-19.
9. C.G. Bell, J.C. Mudge, and J.E. McNamara, *Computer Engineering, A DEC View of Hardware Systems Design,* Digital Press, Burlington, Mass., 1978.
10. J.C. McCallum, "Benchmark Results for Microcomputers and Large Computers," *Data Processing,* Oct. 1986, pp. 426-433.
11. J. Gilbreath and G. Gilbreath, "Eratosthenes Revisited," *Byte,* Jan. 1983, pp. 283-326.
12. H.J. Curnow and B.A. Wichmann, "A Synthetic Benchmark," *The Computer Journal,* Feb. 1976, pp. 43-49.
13. R.P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Comm. ACM,* Oct. 1984, pp. 1013-1030.
14. D.F. Hinnant, "Benchmarking UNIX Systems," *Byte,* Aug. 1984, pp. 132-409.
15. B.A. Wichmann, *Algol 60 Compilation and Assessment,* Academic Press, Orlando, Florida, 1973.
16. B. Randell and L.J. Russell, *ALGOL 60 Implementation,* Academic Press, Orlando, Florida, 1964.
17. C.C. Foster, R.H. Gonter, and E.M. Riseman, "Measures of Op-Code Utilization," *IEEE Trans. Computers,* May 1971, pp. 582-584.
18. "Computerworld's Annual Hardware Roundup," *Computerworld,* Aug. 8, 1983, pp. 31-39.
19. "Hardware Roundup," *Computerworld,* Aug. 19, 1985, pp. 24-34.
20. D. Grune, "Some Statistics on Algol 68 Programs," *Sigplan Notices,* July 1979, pp. 38-46.
21. A.G. Alexander and D.B. Wortmann, "Static and Dynamic Characteristics of XPL Programs," *Computer,* Nov 1975, pp. 41-46.
22. J.L. Elshoff, "A Numerical Profile of Commercial PL/1 Programs," *Software P & E,* 1976, pp. 505-525.
23. A.S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM,* Mar. 1978, pp. 237-246.
24. M.M. Al-Jarrah and I.S. Torsun, "An Empirical Analysis of Cobol Programs," *Software P & E,* 1979, pp. 341-359.
25. I.S. Torsun and M.M. Al-Jarrah, "Dynamic Analysis of Cobol Programs," *Software P & E,* 1981, pp. 949-961.
26. V.S. Foster, "Performance Measurement of a Pascal Compiler," *Sigplan Notices,* June 1980, pp. 34-38.
27. M. Shimasaki et al., "An Analysis of Pascal Programs in Compiler Writing," *Software P & E,* 1980, pp. 149-157.
28. R.P. Cook and I. Lee, "A Contextual Analysis of Pascal Programs," *Software P & E,* 1982, pp. 195-203.
29. G.R. Brookes, I.R. Wilson, and A.M. Addyman, "A Static Analysis of Pascal Program Structures," *Software P & E,* 1982, pp. 959-963.
30. H.J. Saal and Z. Weiss, "Some Properties of APL Programs," *Proc. APL 75,* 1975, pp. 292-297.
31. D.R. Ditzel, "Performance Measurements on a High-Level Language Computer," *Computer,* Aug. 1980, pp. 62-72.

32. S.F. Zeigler and R.P. Weicker, "Ada Language Statistics for the iMAX432 Operating System," *Ada Letters,* May 1983, pp. 63-67.
33. P. Dobbs, "Ada Experience on the Ada Capability Study," *Ada Letters,* May 1983, pp. 59-62.
34. D. Lurie and C. Vandoni, "Statistics for Fortran Identifiers and Scatter Storage Techniques," *Software P&E,* 1973, pp. 171-177.
35. S.K. Robinson and T.S. Torsun, "An Empirical Analysis of Fortran Programs," *The Computer Journal* 1975, pp. 56-62.
36. D. Partridge and E.B. James, "Compiling Techniques to Exploit the Pattern of Language Usage," *Software P & E,* 1976, pp. 527-539.
37. H.W. Whitlock, Jr., "Analysis of the Use of the 6502's Opcodes," *Dr. Dobb's Journal,* Mar. 1981, pp. 11-13.
38. D.W. Clark and H.M. Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780," *Proc. Ninth Annual Symp. Computer Architecture,* IEEE CS Press, Los Alamitos, Calif., 1982, pp. 9-17.
39. D. MacGregor and J. Rubinstein, "A Performance Analysis of MC68020-based Systems," *IEEE Micro,* Dec. 1985, pp. 50-70.
40. M. De Prycker, "On the Development of a Measurement System for High Level Language Program Statistics," *IEEE Trans. Computers,* 1982, pp. 883-891.
41. *Turbo Pascal Version 3.0, Reference Manual,* Borland International, 4585 Scotts Valley Drive, Scotts Valley, CA 95066, 1985.
42. *8080/8085 Assembly Language Programming,* Intel Corporation, Santa Clara, Calif., 1979.
43. *Microsystem Components Handbook,* Vol. 1, Chap. 3, Intel Corporation, Santa Clara, Calif., 1984.
44. *MC68000 16-Bit Microprcessing Unit,* ADI-814-R3 specification document, Motorola Semiconductors Products Inc., 3501 Ed Bluestein Blvd., Austin, Texas, 1982.
45. Motorola Inc., *MC68020 32-Bit Microprocessor User's Manual,* Prentice-Hall, Englewood Cliffs, N. J., 1984.
46. *Eclipse MV/8000 Principles of Operation,* Data General Corporation, 4400 Computer Drive, Westboro, MA 01580, 1980.

John C. McCallum is a senior lecturer at the National University of Singapore in the Department of Information Systems and Computer Science. Before working there, he taught in Canada and in Saudi Arabia. His research interests are in microcomputer applications and in the modeling of information flow, whether in processors or in business organizations.

McCallum obtained his BSc in physics from the University of Western Ontario, Canada, and his PhD in experimental space science from York University in England. He is a senior member of the IEEE.

Tat-Seng Chua joined the Department of Information Systems and Computer Science at the National University of Singapore as a lecturer in 1983. Before that time he worked with the British Gas Corporation in the United Kingdom as part of his PhD thesis to develop software for the simulation of the British Gas Transmission Network. His current research interests include microprocessor systems, visual programming, and the modeling of communication protocols.

Chua received the BSc degree in computer science and civil engineering and the PhD degree in computer science from the University of Leeds, UK, in 1979 and 1983.

Questions concerning this article can be directed to either author at the Department of Information Systems and Computer Science, National University of Singapore, Lower Kent Ridge Road, Singapore 0511.

MicroLaw

Richard H. Stern
Law Offices of Richard H. Stern
2101 L Street NW, Suite 800
Washington, DC 20037

Software copyright developments

Screens and interfaces for computer programs protected by copyright

In the December 1986 issue of *IEEE Micro* we discussed Microstuf's suit against SoftKlone over Crosstalk XVI and Mirror, along with the views of Microstuf's counsel that "look and feel" piracy of that type deserved to be suppressed. On March 31, 1987, the federal district court in Atlanta ruled in favor of Microstuf and permanently enjoined the manufacture and distribution of Mirror so long as it duplicated the main menu of Crosstalk XVI (shown on p. 77 of the December issue). (*Digital Communications Associates, Inc. v. SoftKlone Distributing Corp.*) SoftKlone has indicated that it will revise the main menu of Mirror rather than be forced out of business pending any appeals.

To refresh your memory, Crosstalk is a very successful asynchronous modem communications program for use with IBM PC-type microcomputers. It has a set of 87 commands, settings, and parameters (all of which I refer to as commands) featured in its main menu, and the user is supposed to set them in accordance with the needs of his system.

Because of the number of commands, a uniliteral symbol for each command is not possible. Microstuf therefore adopted a set of biliteral abbreviations for the various commands. For example, **DU** for **DU**plex, **PA** for **PA**rity, **SP** for **SP**eed—other commands are **LO**ad, **PO**rt, **QU**it, **ST**op, **WR**ite, **XM**it. On the status screen, the capital letters shown here in boldface are displayed in high-intensity monochrome, and the user understands that the high-intensity biliteral is the code for the whole word. To set a parameter, such as speed, the user enters the biliteral and the correct speed; for example, for a rate of 1200 baud, the user enters SP1200.

SoftKlone decided to market an emulator using the same status screen, apparently on the theory that the public is used to Crosstalk and is unwilling to learn to use a different user interface. The court held that to be copyright infringement.

What are the implications for software innovators and emulators? At least superficially, the court seems to want to close the door on emulators, by forcing them to develop different-looking user interfaces. Therefore, whoever first appropriates the most logical and convenient user interface for a particular function would seem to be able to prevent latecomers from following suit. The court seems to believe that it was not doing that, but its belief rests on erroneous suppositions about how user interfaces for microcomputer software work and about the range of options available to designers of menus for microcomputer software.

At the outset, the court refused to consider that DCA's copyright in the computer program covered the screens. However, DCA had registered a separate copyright claim for the screen as a compilation of parameters and commands arranged on the status screen in an original manner. This was, for obscure reasons, classified as a "literary work," although the court protected the graphic and visual aspects of the status screen, rather than its literary content. The court found the "compilation" to be copyrightable and to have been infringed.

The court began its analysis with a statement of the customary idea/expression dichotomy, noting that particular expressions of ideas are protected under the copyright laws. But ideas, as such, are not so protected. The court then applied the principle to the case before it by finding that each of the following is an idea: the use of a screen to reflect the status of a program, the use of the command-driven program, and the use of biliterals to activate a command.

On the other hand, the court found that the arrangement of biliterals on the screen was an expression, because in this program the order in which the user enters the commands makes no difference to the operation of the program. For example, whether **PA**rity is entered before or after **SP**eed makes no difference. Therefore, SoftKlone could have shuffled the rows and columns of commands on the status screen without impairing the program's utility.

Even more important, the court found that "the highlighting and capitalizing of two specific letters of the parameter/command...has no relation to how the status screen functions," and therefore that is expression rather than idea. This is the point at which I think the court fell down, and the part of the ruling that may create the most uncertainty and difficulty for writers of microcomputer software. The court asserted:

> The defendants could have used a wide variety of techniques to indicate which

> symbols the user should type to effectuate a command, e.g., different symbols could have been chosen, or simply highlighting, or capitalizing, or underlining the appropriate symbols, or any combination thereof, or placement of the symbols in parentheses or square brackets before or after the parameter/command. The modes of expression chosen by the plaintiff for its status screen are clearly not necessary to the idea of the status screen. Therefore, the plaintiff's mode of expression of the status screen does not merge with the idea of the status screen.

That is to say, the SoftKlone menu designer should not have used the format **SP**eed to represent that command, but should instead have used one of the following: speed, Speed (SP), [SP] Speed, speed, SpeEd, SpeeD, s**P**Eed, and so on. (We might add these too: SP—speed, SPEED—sp, speed: SP, speed = SP, speed...SP.)

That is simply foolish. Only four screen attributes exist that are available for a monochrome PC-type microcomputer: high-intensity video, inverse video, underline, and blinking. That entirely exhausts the options, besides capital and lowercase letters. The fourth attribute, blinking, is useless. No one in his right mind would use that for a status screen; it would antagonize any user and probably give the user a headache as well. The underline attribute is almost as useless. It clutters up a screen, and it is not very effective in letting letters stand out from the rest of the text on the screen. That leaves two usable attributes, high-intensity and inverse video.

The use of high-intensity video in an all-lowercase word, in my opinion, does not sufficiently emphasize the selected letters to make them easily and comfortably recognized by the user. We could try to verify that, I suppose, by giving people flash card tests, but I believe readers will agree with me either on the basis of their reading the boldface text two paragraphs back or after trying it out on their own microcomputers. Therefore, I believe that it is necessary to combine high-intensity video with caps for a good user interface. The same is true, also, I believe, for inverse video, although perhaps not to quite the same extent. It therefore begins to appear that the infinite range of expressions available to menu writers, which the court imagined, is actually limited to something much smaller—involving the use of inverse video with, at best, either caps or lowercase letters (and perhaps only caps), and high-intensity video with caps.

Let us turn to the other dimensions that the court thought made for infinite possibilities. Examination of the examples shown above for use of other than the first two letters will, I believe, convince any unbiased viewer of the wrongness of using other than the first two letters of a command or other keyword. It confuses the user and looks silly. It is simply nonintuitive and not good practice to design a user interface that way. Finally, the use of parentheses, square brackets, and dashes to delineate the command biliteral is slightly confusing and at any rate takes more space and clutters up the screen. The use of equal signs may not be confusing, but it does take up more room than simply emphasizing the selected biliteral.

This use of copyright law throttles competitors and users. It is getting very badly out of hand.

In short, we see that the best format for a user interface of this type—or at least, one of the three best—is preempted by copyright law, as interpreted by the court. The implication is that whoever first appropriates the most desirable format for a user interface can now prevent competitors from using that format when they set out to imitate the originator's software, even though they write their own code. And after another one or two firms enter the market, all the usable interfaces will be preempted.

Presumably, enthusiam for that notion was responsible for Lotus' suing Paperback Software and Mosaic Software in January 1987 for misappropriating the look and feel of Lotus 1-2-3, and for VisiCalc's creators' (SAPC) suing Lotus in April 1987 for the same alleged rip-off of VisiCalc's look and feel.

Where VisiCalc took the look and feel from, I cannot imagine. (That may be the next lawsuit.) But, dollars to doughnuts, the overwhelming majority of users feels like this:

> I expect to be able, without hindrance from the legal profession or its greedy clients, to enter the first letter or first two letters of a command to invoke it, or else move the cursor to the command and press Enter, on any and all programs of the spreadsheet, database, and similar types. I also expect to enter F1 for help, ESC or backslash for escape from present screen, and slash for command mode; and I am going to be very unhappy if anybody expects me to learn something different. I have a number of similar expectations, and I do not want to be forced to change them because of the copyright laws. It is hard enough to learn how to use these tools, without having to learn to use a new kind of interface every time, and I have preferable ways to spend my time (such as drinking beer and watching TV).

This use of copyright law to throttle competitors, and also the user community, is ridiculous. And it is getting very badly out of hand. I have a modest proposal for action by the Computer Society of the IEEE.

A standards committee (or if it is too hard and time-consuming to go through the IEEE bureaucracy, then a Software Users Defense Committee) should establish criteria for good user interfaces for microcomputer programs. Part of this effort (or all of it) should be criteria for proper menus or screens or a set of them. For example, where there are only a few commands on a screen, the equal sign or a box with vertical lines separating uniliteral commands from their explanations will probably be considered an acceptable alternative. But for crowded screens, it may well be that we will all agree that high-intensity video and caps for the first one or two letters of a command or keyword is the best option, perhaps along with inverse video for some uses.

Without going into the details of how to prescribe the end result, my point is that I would like to see an agreed-upon approach to user interfaces. Then, I would like to see it put into the public domain for the benefit of users of computer programs, who do not want to use unfriendly interfaces or learn to use many different ones, and who would be happiest if they kept seeing the same old interface all the time.

The idea would be that anyone who stuck to the IEEE interface would be immune from harassment under the copyright laws. Say previous rights under the copyright laws were asserted to some technique that was part of the IEEE-

Continued on p. 89

MicroReview

Richard Mateosian
2919 Forest Avenue
Berkeley, CA 94705
(415) 540-7745

In my April column I made some rash statements about what I was going to do this month. I'm not going to do any of them. Instead, I've reviewed two books that are similar to one another in important ways. One deals with CD ROMs and the other with desktop publishing, two rapidly growing areas for small computer applications. Each is based on a relatively new hardware technology that is quickly dropping in price into the consumer product range.

The CD ROM book is written for developers by developers, the other is written for end users by end users. Each aims at giving the reader an orientation to the given field. Both books were produced quickly in an effort to bring out in book form information of the currency usually found only in periodicals. Both show the bad effects of this haste, although I am not aware of factual errors in either of them.

CD ROM 2: Optical Publishing, ed. by Suzanne Ropiequet with John Einberger and Bill Zoellick (Microsoft Press, Redmond, Wash., 1987, 384 pp., $22.95)

Imagine 500 megabytes on a mass-produced disk you can carry around in your pocket and access with consumer-grade hardware! It's hard to grasp at first, but it soon hits you that this is a technology that you ought to find out about.

The subtitle of this book is "A Practical Approach to Developing CD ROM Applications," and that's a pretty good description. What's a CD ROM? Here's the CD part, from the book's glossary:

> Compact Disc: The trademark name for an injection-molded aluminized disc, 12 cm in diameter, which stores high-density digital data in microscopic pits that a laser beam can read. Conceived by Philips and Sony, it was originally designed to store high-fidelity music for which Compact Disc Digital Audio now is a standard format accepted worldwide. Because of its very large data storage capacity, the Compact Disc now is used as a text/data medium in electronic publishing (CD ROM).

The physical format used for storing general-purpose digital data for personal computers on CDs is established in the CD ROM standard, also known as the Yellow Book. The de facto logical format standard is the *High Sierra Group Proposal,* the work of a group of people from Apple, DEC, Hitachi, Microsoft, 3M, Philips, Sony, and others. Under these standards, a CD ROM contains 270,000 sectors of 2048 data bytes each, or a total of more than 540 megabytes.

The most interesting CD ROM applications probably haven't been thought of yet, but a typical current application is an electronically published encyclopedia with full support for browsing and for following references from section to section. Potential developers of applications like this will find that the book is an excellent introduction and reference, but there is one caveat before we proceed with the details. The publisher is a division of Microsoft Corporation, an active participant in the development of this field, so you may not be getting total objectivity.

The book is a collection of 16 essays, mostly by different authors, covering just about anything that a prospective developer of CD ROM applications might want to know. The authors are experts in their fields, and each chapter is well organized and full of interesting material. I recommend reading this book cover to cover, then keeping it around for reference.

Actually, the book starts off pretty badly. Page one contains a typographical error, an example of sloppy editing, and an absurd statement. In fact, if you're sensitive to this sort of thing, you ought to start with Chapter 2. The editing never gets much better, but the typos taper off and the contents become a lot more substantial. (Having complained so pointedly about the editing, I suppose I ought to give an example. My favorite is the statement that certain techniques "...are never perfect—almost always retrieving far more irrelevant data than you need.")

The four chapters on text preparation and retrieval make up the most issue-oriented section of the book and one of the most interesting. How, for example, can existing textual material be transferred to CD ROM in such a way that its structure is visible to retrieval software? Even though a huge proportion of everything published today is originally produced on computers, many barriers exist to accessing even the text of those originals; recovering the structure without prohibitively expensive human intervention is usually impossible. One answer suggested in the book is to use the *Standard for Electronic Manuscript Preparation and Markup* devised by the Association of American Publishers. Widespread use of the AAP standard, it is suggested, will depend upon the availability of word processing software that supports it. Is this a hint of Microsoft's future plans?

Another interesting issue is searching versus browsing as approaches to document retrieval. Searching, the "standard" approach, is shown to have severe problems. There seems to be an inverse relation between the two key measures of searching effectiveness: completeness and relevance. The larger the list of documents retrieved using a given search key, the lower their average relevance to the user's problem. The higher the average relevance of the documents retrieved, the greater the proportion of relevant documents that will not be retrieved. Browsing, on the other hand, is more natural and more effective, but it requires the system to know the structure of documents, not just their text.

These issues are not new, but in the past they have had to be faced only in large systems. With CD ROMs far more system designers will need to deal with

these issues, and they will be doing so for applications of a different scale, in which the cost/benefit analyses of the various approaches will depend upon different parameters.

Another problem that is not new is selecting the right index structures for huge text databases. But the problem is given a new twist in the CD ROM environment. The read-only nature of CD ROMs, their large capacities, and their slow access times are all factors that influence the selection of index structures. For example, a lookup that requires six disk accesses may be perfectly acceptable for a standard hard disk but painfully slow on a CD ROM.

Another interesting section of the book provides a programmer's view of graphics and audio. Many books talk about these subjects, but this one seems to strike just the right balance between brevity and thoroughness. The reader is assumed to be intelligent and generally sophisticated about computer technology and applications, but is not assumed to have a background in electronics or to know much about graphics or audio.

If you decide to develop applications for CD ROM, you will want to obtain and study the *High Sierra Group Proposal,* currently on its way to becoming an international standard. A chapter of the book contains a description of the proposed format and the issues behind it, written by two members of the High Sierra Group. The proposal defines a full (Level Three) implementation, which provides essentially all the needs of the various sponsoring organizations.

Two standard subsets are specified. Level One is for minimal systems, and Level Two is slightly augmented to provide for compatibility with CD-I. (CD-I is a controversial proposed standard hardware/software environment for consumer products to be delivered beginning in 1988. This book contains little mention of CD-I and no mention of the controversy surrounding it.)

Two related subjects are the protection and updating of the data in CD ROM products. Those of you who regularly read Richard Stern's MicroLaw columns will be familiar with many of the legal issues in the area of protection of intellectual property, but the discussion in this book focuses on the specific problems raised by the nature of CD ROMs. For example, the doctrine of first sale gives the purchaser of a work the right to display it publicly, while the copyright holder retains the right of performance. What these terms mean for a CD ROM database, possibly containing music and images, is not clear. If you're going to develop CD ROM applications, you'll want to know a lot more about issues like these.

While updating of CD ROM databases has legal ramifications, the software issues are even more interesting. The logical format embodied in the *High Sierra Group Proposal* makes it possible for one CD ROM of a multiple CD ROM set to alter the interpretation of data on other CD ROMs of the set. This makes it possible for an updating CD ROM in effect to delete or modify data previously supplied. Thus, a database might be supplied on five CD ROMs, and periodic updating of only the fifth CD ROM could effectively update the entire set.

The book also contains practical advice for potential developers. While the one-minute business plan contained in Chapter 2 isn't worth much, there are practical, advice-filled chapters on disk origination and mastering, and there are two interesting and instructive case studies. And there's an appendix called "Resources," which contains classified listings of firms involved in the CD ROM field.

If you read *IEEE Micro* and you don't already have a pretty good grasp of the subjects in this book, then it's a "must read" for you.

The Art of Desktop Publishing, 2nd ed., Tony Bove, Cheryl Rhodes, and Wes Thomas (Bantam, Toronto & New York, 320 pp., $19.95; $24.95 in Canada)

This book is subtitled "Using Personal Computers to Publish It Yourself," and because that's exactly what the authors have done, you can get a good idea of the pros and cons of being your own publisher. The authors, by virtue of their experience with this and other publishing projects, can help to guide your steps along this path, and the book, as a sample product, can teach you lessons that the authors didn't make explicit.

The first question you should ask yourself when considering a publishing project is "What do I hope to accomplish?" If your answer is along the lines of getting your message out quickly and correctly, then personal publishing is worth considering. For example, the authors produced this second edition in two weeks. If, on the other hand, your answer has a heavy component of impressing the reader with the quality of the result, you'd better think seriously about getting professional publishing help. In this respect, a publishing project is a lot like a hardware or software engineering project.

This book is written by three people who are far from amateurs in publishing, but the most generous grade I can give it as an example of publishing is B-, and an even lower grade could easily be justified. On the other hand, I enjoyed reading the book and found parts of it to be useful and informative. It has the flavor of a collection of trade-press articles and newsletter excerpts wedded to tutorial articles on page makeup programs (mostly Pagemaker).

This makes a convenient package for someone who doesn't follow the trade press, doesn't subscribe to a newsletter, and can't learn enough from the manuals accompanying the page makeup programs.

At this point I should say that Aldus Corporation shipped me Pagemaker for the IBM PC when it came out in February. I don't know if this was true in the past, but the manuals that accompanied that shipment don't seem to need to be supplemented by outside tutorials. They're full of tutorial information and production advice.

To pull all of the above pros and cons into a recommendation, I'd say that there are many people who will want or need to learn more about personal publishing and to purchase software and equipment. If you're one of these people and you feel a little bewildered by all you've heard about the "desktop publishing revolution," this book can help.

Next time

As noted earlier, last issue's prediction of future plans proved to be completely incorrect, largely because I simply didn't get far enough in reading Maurice Bach's *Design of the UNIX (TM) Operating System.* I hope to finish that book and to look at other interesting books and software for the August issue.

MicroStandards

Michael Smolin
Smolin & Associates
3428 Greer Road
Palo Alto, CA 94303

At the March 1987 meeting of the IEEE Standards Board several new standards were approved. Among them were:

- 802.5A, LAN: Station Management (supplement to IEEE Std. 802.5-1985),
- 854, A Radix Independent Standard for Floating-Point Arithmetic,
- 1014, Specification for a Versatile Backplane Bus (VME), and
- 1016, Software Design Descriptions, Recommended Practice.

A project newly authorized at that meeting was P1141, Forth: A Microcomputer Language Standard. This project is likely to become a joint project with X3.

Four backplane bus standards projects sponsored by the Technical Committee on Microprocessors and Microcomputers, or TCMM, have passed their sponsor ballots. They have been submitted to the IEEE Standards Board for adoption as IEEE standards. These projects are:

- P896.1, the Futurebus Backplane,
- P1101, the Mechanical Core Specifications for Microcomputers,
- P1196, A Simple 32-bit Backplane Bus (Nubus), and
- P1296, A High Performance Synchronous 32-bit Backplane Bus (Multibus II).

Seven new project authorizations are being requested of the IEEE Standards Board by the Computer Society's TCMM. These projects are:

- P1151, Modula II, A Modular High Level Programming Language,
- P1152, Smalltalk, An Object Oriented Programming Language and Environment,
- P1153, Page Descriptor Language,
- P1154, PILOT, A Program Instruction Learning or Teaching Language,
- P1155, A High Speed Backplane Instrumentation Bus,
- P1156, Connectors and Mechanical Packaging for High Reliability Bus Structures, and,
- P1496, Rugged Bus, A Very High Reliability Bus Structure.

In addition, a project authorization has been requested for the revision of *IEEE 755, Extending High-Level Language Implementations for Microprocessors,* a trial-use standard. This has been a contentious standard. It has already passed an appeal against its adoption that was filed by The Pascal Joint Committee chairman.

If these project requests are approved, I will include the details of the scope and list the chairman of each project in the next issue of *IEEE Micro.*

On another note, James (Bob) Davis, the chairman of the Microprocessor Standards Committee, has appointed Paul Borrill to be the chairman of P896.2, the project to develop a Futurebus Firmware Standard. Borrill can be reached at:

Spectra Tek US Ltd.
Swinton Grange
Malton
North Yorkshire Y017 OQR
England
Telephone: (0653) 5551.

MicroNews

MicroNews features information of interest to professionals in the microcomputer/microprocessor industry. Send information for inclusion in MicroNews one month before cover date to Managing Editor, IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.

Materials capture industry's imagination

Materials don't seem to be the subject to stir the imaginations of most of us. After all, what can be new and interesting about such a standard topic?

But a great deal of recent news has centered on the changes in the materials used by the electronics industry to produce commercial and military devices. These changes have been so dramatic in one case at least that we find ourselves speculating about the materials of the future.

Time magazine devoted its May 4, 1987, cover to the multitude of effects that the recent advances in superconducting materials will have on our world. Another interesting material, gallium arsenide, though no longer brand new to the market, still inspires electronics manufacturers who hope to capture its unusual qualities to produce better devices. And just now reaching the commercial market are chips based on diamond thin film.

These materials are widening our horizons in more ways than one; here's a quick look at some of the research and commercial plans.

Superconductors

When can we expect to see superfast computers?

Very, very soon would seem to be the answer if current research succeeds in being applied.

Recent physics advances in high-temperature superconducting materials have been making headlines as companies, government agencies, and universities around the world race to experiment in transmitting electricity with little loss of energy. Superconducting materials lose all resistance to electricity below a specific temperature, a quality very likely to produce much faster electronic devices and thin films.

The larger energy gaps experienced in these materials occur 10 times more often than do those in present superconducting integrated circuits. This energy increase means faster devices can be produced; it also suggests that the physics of these materials may be very different from that of conventional superconductors.

The superconductor race. Discovered in 1911 and advanced slowly over the years, superconductivity only recently became the object of concentrated research. Four significant achievements occurred in 1986. Early in the year K. Alex Mueller and J. Georg Bednorz of the IBM Zurich Research Laboratory reported that a ceramic containing lanthanum, barium, copper, and oxygen showed traces of superconductivity at 30 degrees kelvin. (The Kelvin scale starts at a temperature of absolute zero, the point at which all motion of atoms ceases.) Before this discovery, the best commercially available superconductors were cooled to 23.2 K by bathing them in $5-a-liter liquid helium.

Later that year scientist Shoji Tanaka at the University of Tokyo reported a structure for the compound, convincing others of the reality of the discovery.

Last December physicists Robert J. Cava of AT&T Bell Laboratories and Paul C. W. Chu of the University of Houston discovered superconductivity at 36 and 40.2 K. That same month Z. X. Zhao at the Chinese Academy of Sciences reported success at 44 K, and in February Chu repeated his achievement, this time at 93 K with yttrium barium copper oxide. At 77 K it is possible to use the more-readily available, 10-cent-a-liter liquid nitrogen to cool materials.

Since February, scientific investigators at the Argonne National Laboratory have found at least 13 other ceramics that are also superconducting at temperatures between 90 and 95 K. Argonne crystallographers have determined the structure of the material discovered by Chu. These scientists precisely located the oxygen atoms in the crystals with the help of equipment called the Intense Pulsed Neutron Source. Information about the structure should give hints about other types of materials that might be superconducting.

While it took scientists 75 years to raise superconductivity temperatures by 19 degrees, it took only a little over a year to raise it from 23 to 95 K.

But what's happened lately? Applications of the new superconductors might include their use as interconnects in semiconductor computers at liquid nitrogen temperatures. Pattern microstructures have been obtained and are currently being tested at Stanford's Department of Applied Physics.

Argonne National Laboratories has been producing thin films, pellets, and 0.006-inch wires from the new material and measuring their characteristics. Scientists at AT&T Bell Laboratories in

Murray Hill, New Jersey, reported that they also have been able to make wires flexible enough to be wound into coils. These are key steps in readying the materials for commercialization.

Just recently, IBM announced a thin-film superconducting device based on copper-oxide material. The junction devices measure one one-hundredth the thickness of a human hair and are called Superconducting Quantum Interference Devices. SQUIDs are chilled by liquid nitrogen.

An even more spectacular IBM announcement concerned its success in increasing the current-carrying capacity of superconductors by 100 fold. IBM scientists grew a one-inch-diameter thin-film single crystal and cooled it below 77 K. The crystal carried a current of 100,000 amperes per square centimeter. When the scientsts cooled the thin film to near absolute zero, they found that it conducted 5 million A/cm^2.

What's waiting in the future? Well, researchers at Wayne State University recently announced they had seen evidence of superconductivity at 240 K. And, Paul Chu foresees superconductivity at 300 K (room temperature) eventually. And, with IBM's current-carrying advances, we can expect to see widespread use of these materials in the variety of applications promised us on the cover of *Time*. It seems we'll all be winners in the superconductor race.

GaAs

Another material continuously generating industry interest because of its performance advantages is gallium arsenide. Two agreements have joined Rockwell International with IBM and Honeywell with the Defense Advance Research Project Agency in the pursuit of GaAs technology advances.

The Rockwell International/IBM agreement calls for cooperative development of advanced gallium arsenide technology and production techniques. Their effort will concentrate on developing cost-effective optoelectronic and digital components needed for special uses in computers and telecommunications equipment of the future.

The program involves development teams at Rockwell's California facilities in Newbury Park and Thousand Oaks and at IBM's New York facilities in East Fishkill and Yorktown Heights.

The Honeywell/DARPA contract has recently produced and demonstrated an integrated GaAs monolithic receiver chip at 1-gigabit clock frequencies. The second-generation chip contains a photodetector and 200 gates on the same GaAs substrate. It is useful for optical interconnects between computer chips and from computer to computer.

According to Honeywell, the 2mm × 2mm device is compatible with current manufacturing processes using direct ion implantation MESFET technology and metal organic chemical vapor deposition for epitaxial growth. The receiver contains an optical detector, preamplifier circuit, and 1:4 demultiplexer. It was designed to decode a 1-gigabit optical signal input into four parallel 250M-bit electrical outputs.

Meanwhile, help in production control has come from the National Bureau of Standards, which recently developed two polarized infrared light systems designed to detect flaws in GaAs semiconductor materials. Both infrared systems are nondestructive methods that wafer manufacturers can use to screen materials before marketing. One system examines an entire wafer, while the other employs a 75- to 600-X microscope to view isolated wafer portions.

Both systems digitally store images and use false-color graphics to transmit infrared intensity, which could indicate potential problems. Bureau researchers use the techniques and will assist businesses in setting up their own systems.

Diamond thin film

Research and development into DTF-based chips is progressing to the commercial market in Japan, according to a recent International Resource Development report. Shinetsu Chemical Company is shipping diamond film-coated knives for electron microscopy, and Sony is marketing a loudspeaker tweeter that uses the material. Sumitomo is expected to soon be releasing its first DFT-based chips for applications involving hostile environmental conditions, such as in spacecraft or automobile engines.

Diamond film possesses unique mechanical, electronic, and optical properties, which have applicability in a wide range of military and commercial markets. For example, it seems that DTF chips will be superior in speed and in environmental resistance properties to gallium arsenide.

Despite early research at Case Western Reserve in the US, it was researchers in Moscow who in 1977 came up with some key insights into how to manufacture synthetic diamond in thin-film form, using chemical vapor deposition techniques. Japan, the USSR, and the US competed to find commercially viable manufacturing processes for the new material.

Today, the Japanese are in the lead, with 1987 shipments expected to total $17 million. Industry research reports predict the $400-million level will be approached by 1993.

Will we soon replace the Fourier Transform with the Hartley Transform?

A native Australian working at Stanford University has invented an algorithm to replace the famous Fourier Transform and is trying to build a chip containing the algorithm. Called the Hartley Transform by inventor Ronald Bracewell, the new equation cuts in half the amount of time needed to perform the same mathematical analysis, uses half the computer memory, and resides on a much smaller or lighter chip than those containing the Fourier equation.

Bracewell first became fascinated with the Fourier Transform in school in 1940. He lectured on Fourier analysis in 1955 and published a book containing Fourier's work in 1965. "It has sort of permeated my whole life, you might say," comments Bracewell.

Five years ago Bracewell decided to write down his thoughts about ways of possibly improving on Fourier; he based his idea on the work of Ralph Hartley done at Bell Laboratories in 1942. The result was a fast algorithm—developed, as Bracewell says, with "...some of the hardest mind grinding I ever did. It took me months and months, straining my brain."

Mindful of the need to patent the equation, Bracewell is trying to convince other engineers to build a chip containing the algorithm so it will qualify under the law as having a physical presence. Taiwan, he has heard, is presently developing just such a chip.

New staff member

Assistant Editor Christine Miller joins *IEEE Micro* after assignment with both *IEEE Expert* and *Design & Test* magazines. She holds a BA in English from California State College at Fullerton, has taught English and has authored some 25 feature magazine articles, including a series on air and water pollution. In addition to previous magazine editorial experience, she is editor of a financial planning book to be published this summer.

Her interests in addition to literature are science, drama, music, and the art of conversation.

Christine is excited about working on our staff and welcomes your communiques.

China's computer imports to reach $3.5 billion

The western nations will export 4200 minicomputers and mainframes and 160,000 microcomputers to The People's Republic of China in the next five years, predicts an International Data Group announcement. This represents a total import value of $3.5 billion in computers and related technology.

IDG, based in Framingham, Massachusetts, has direct experience doing business in the P.R.C. It publishes the biweekly newspaper, *China Computerworld*, which is headquartered in Beijing, has 100,000 paid subscribers, and is said to be read by two million people.

DEC-compatible controller guide

Dilog is offering a product guide to its DEC-compatible peripheral and communications controllers. The guide has been designed as an exact-size replica of a dual-size disk drive controller, complete with die-cut edge connectors and embossed ICs. The guide provides information on products for use with MicroVAX, PDP-11, LSI- 11, and VAX Unibus computers as well as listings of all disk and tape drives that are compatible with the company's controllers.

For a free copy write to Dilog Product Guide, PO Box 6270, Anaheim, CA 92806.

$4-billion market projected for 80386-related products

Computers, software, and peripherals supporting the Intel 80386 32-bit microprocessor should top $4 billion in 1991 and level out to about $3.4 billion by 1993, according to recent research findings. The increased computing power of the chip promises higher speed, multitasking, and large memory access, advantages avidly sought by users.

Areas expected to be affected by the 32-bit computers are the CAD/CAE and office automation markets now served by supermicros, minicomputers, and mainframes. The report cites the key issues of standardization, IBM's marketplace, and operating systems as crucial in planning market strategy.

Markets for Products Based on the Intel 80386 Microprocessor: Systems, Software, and Peripherals can be purchased for $995 from Market Intelligence Research Company, 4000 Middlefield Road, Palo Alto, CA 94303; (415) 856-8200.

Museum offers early PC slides

Collecting unique relics of the personal computer revolution is becoming easier for hobbyists and other interested parties.

Now, a color slide series of PC images can be obtained from The Computer Museum in Boston. Twenty images of the first personal computers, hobbyist milestones, homebrew and single-board computers, and early and classic commercial machines can be purchased for $20. Volume I of the series, available for $45, contains 48 slides of early calculating devices and computers, supercomputers, logic and memory technologies, and classic integrated circuits.

Write to The Computer Museum Store, 300 Congress Street, Boston, MA 02210, to order either volume. Please add $2.50 to cover postage and handling charges.

Current literature

National Semiconductor Corporation is providing customers with a real-time electronic catalog of RETS military test specifications for ICs, which can be accessed by company sales personnel in the US. The directory includes a listing of the electrical tests performed on all military devices qualified by the company and a history of test-program revisions.

National Semiconductor Corporation, PO Box 58090, Santa Clara, CA 95052-8090; (408) 721-5407.

A three-tape audiocassette course, "An Introduction to the MC68020 32-Bit Microprocessor," discusses the major enhancements of the Motorola device over the original MC68000. Course notes, user's manual, and related literature support the self-paced tapes.

Motorola Semiconductor Products Sector Technical Operations, PO Box 52073, Phoenix, AZ 85072; (800) 521-6274; $95.

The design and test of a 16-bit computer system around the Motorola MC68000 is the aim of this 592-page text from Bucknell University's College of Engineering. Author Alan D. Wilcox integrates the principles of engineering with practical hands-on experience.

Prentice-Hall, Englewood Cliffs, NJ 07632; (800) 526-0485; $42.95.

Memory Discontinued Devices displays specifications, logic drawings, and pinout information for 15,500 memory ICs previously available from 92 manufacturers but no longer in production. Devices covered include RAMs, ROMs, PROMs, EPROMs, EEPROMs, programmable logic ICs, code converters, European- and Asian-character generators, and bubble memories.

D.A.T.A., Inc., 9889 Willow Creek Road, San Diego, CA 92126; (800) 854-7030; in California, (800) 421-0159; $65.

Infonet, Inc., is publishing the *Japan Computer Index '87*, an English-lan-

guage hardware/software directory of the Japanese computer industry. Over 5000 company listings, analyses, reviews, and projections appear.

Infonet, Inc., 5F The 7th Industry Bldg., 1-20-14 Jinnan, Shibuya-ku, Tokyo, Japan 150; (03) 770-4483; software, US$280; hardware, US$215.

Information about IEEE-488 bus interface (GPIB) products for IBM PCs and compatibles, Apple, AT&T, Tandy, Texas Instruments, Apollo, Sun, Compaq, Motorola, and NCR appears in the 24-page 1987 catalog published by National Instruments.

National Instruments, 12109 Technology Boulevard, Austin, TX 78727-6204; (800) 531-4742; in Texas (800) IEEE-488; free.

The biweekly *Superconductors Update* is a printed current-awareness subscription service containing abstracts of superconductor research and bibliographic citations from patent documents, journals, and other publicly available literature. The service includes a 650-page book, *Superconductors Update, January-March 1987,* and biweekly updates.

STN International, 2540 Olentangy River Road, Columbus, Ohio 43202; $750.

Analogic Corporation is offering its *Data Conversion Systems Digest* without charge to chief engineers and system designers. The compendium supplies practical tutorial and reference material that addresses typical design problems encountered by engineers. Topics include A/D conversion architectures, system applications, and ground loops and interference.

Analogic Sales Administration Office, 8 Centennial Drive, M/S 5E7, Peabody, MA 01961; (617) 246-0300.

Three recent books from Meckler Publishing offer guides to CD ROMs. *Publishing with CD-ROM* written by Patti Myers explores compact disc optical storage technologies for providers of publishing services ($19.95). *CD-ROM and Optical Publishing Systems* by Tony Hendley assesses the impact of optical read-only memory systems on the information industry and compares them with traditional publishing systems ($39.50). The *Guide to CD-ROMs in Print* is an annual reference book using the books-in-print concept to list currently available CD ROMs and other digitally encoded optical medium products ($29.95).

Meckler Publishing, 11 Ferry Lane West, Westport, CT 06880; (203) 226-6967.

A new monthly publication edited by Charles Rolander, *Electronics Industry Outlook*, tracks business trends in the electronics industry. The report features analyses and forecasts using a top-down approach for continuity and computer-generated charts and graphs for readability.

HTE Management Partners, 4575 Scotts Valley Drive, Suite 105, Scotts Valley, CA 95066; (408) 438-2395; price not stated.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 183 Medium 184 Low 185

MicroLaw

Continued from p. 82

recommended interface. In this case, it should be a Computer Society project to do the research needed to show either (1) that someone else did it earlier than the claimant of alleged previous rights, or (2) that functional considerations about a proper interface make the technique functional and utilitarian, and thus an idea rather than an expression.

If the software publishers had any common sense, they would join us in devising the standard. As Dan Bricklin, a cocreator of VisiCalc, warned at a recent Massachusetts Computer Software Council roundtable: "A lot of companies are going to rise and fall because some lawyer will be able to pull off a court trick. In this climate, if I were an investor, I'd be afraid to invest in any software company."

Or, as Cullinet Software's former president John Cullinane put it: "It's going to have a deadening effect in regard to innovation. Because of the legal uncertainty about look and feel, the issue tends to inhibit dynamic, small, underfunded organizations from developing something better. These days, it's very important that you retain very good counsel." He has several good points, there. I do not want to knock retaining very good counsel—that is a first-class, highly recommended idea. But it would be considerably cheaper just to devise a good industry-standard interface and adhere to it as insurance and immunization against copyright infringement litigation over your interface.

Then we could concentrate on litigating more important and profound copyright issues, such as those past favorites—who really should own the exclusive rights to the desktop metaphor and the trashcan icon or whether copyright should protect the logic blown into a programmed field-programmable device.

In a future issue I will fill you in on a new wrinkle in audiovisual work copyrights—how copyright protects the gestures of mechanical toys like Teddy Ruxpin, and how that theory can be applied to protect printed circuit boards for a mechanical parrot vending machine.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 171 Medium 172 Low 173

New Products

Marlin H. Mickle
University of Pittsburgh

Send announcements of new microcomputer and microprocessor products, and products for review, to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

Dial-back system prevents unauthorized access

A security unit from Britain's GEC Telecommunications Ltd. prevents unauthorized access to a computer over standard PTT lines—even when an intruder enters a legitimate user name and password—by discontinuing the connection and redialing the caller.

When a call is received, the DSU 0496 dial-back security unit responds with a welcome message, prompts for a user name and password, and breaks the connection. The name and password given by the caller are then checked against an authorized list stored in memory; if found to be legitimate, the caller is contacted by the unit on a telephone number that has been approved for that particular user. Computer connection is limited to programmed telephone numbers.

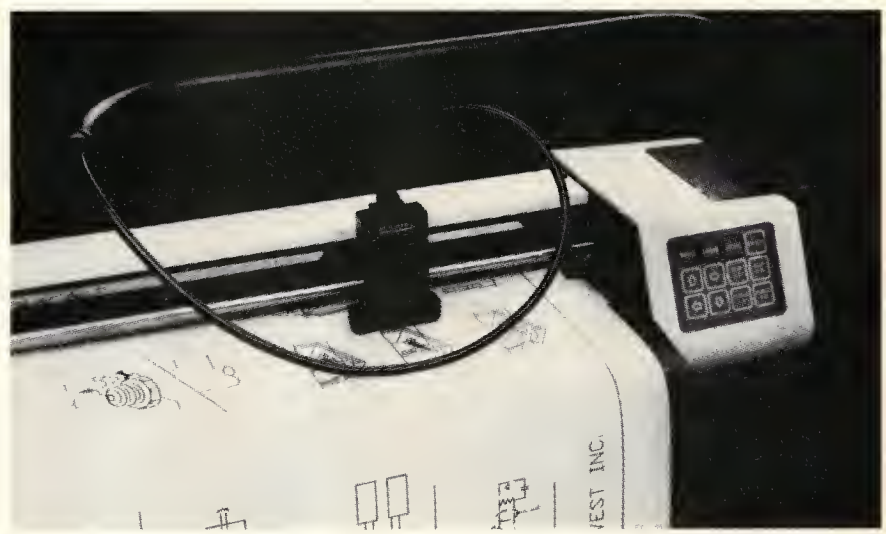
Managers can enter names, passwords, and telephone numbers for 200 users by accessing the system's editor program with a security lock and key. Editors also control all line speeds and characteristics, which can be set independently for each modem and computer connection. The unit converts speed and format between the modem and computer.

A printer can be connected to the unit to provide user access times and call durations for call charging and for logging unauthorized access attempts. One unit simultaneously secures four dial-in access ports. Modems up to 9600 bps are acceptable.

Contact GEC Telecommunications Ltd. for pricing.

Reader Service Number 30

Scanning input device aids pen plotters



With the Scan-CAD accessory, Houston Instrument DMP-50 series plotters can be converted into an automatic digitizer. As a drawing is scanned, the hard-copy image is converted by the scanning software into a raster data file, which can be used as is or read by other software for further conversion into vector data.

Houston Instrument has announced a scanning input device as an option for its DMP-50 series pen plotters. The Scan-CAD plotter accessory features a 200-dpi scan head that detects lines of 0.007 inch and automatically scans detailed architectural, engineering, or other CAD drawings from paper, vellum, acetate film, or blueline stock.

According to the company, when using an IBM PC AT with a drawing of medium complexity and a scan velocity of 2 ips, Scan-CAD can input a D-size drawing in 12 minutes and an E-size drawing in 24 minutes. Scan-CAD includes a snap-on scan head, cable and cable support assembly, scanner controller expansion card, scanning software, and operation manual. The unit requires an IBM PC or AT with 10M-byte or larger hard disk and 640K memory and a Houston Instrument DMP-50 series drafting plotter.

Priced at $2995, the plotter option is expected to be available in December.

Reader Service Number 31

VAX-version DSP adapted for IBM PCs

Signal Technology Inc. has announced that Version 6.0 of its Interactive Laboratory System allows IBM PC users to access the range of ILS programs formerly only available to VAX/VMS users. The PC digital signal/speech processor includes a menu-based user interface, color graphics, and additional data acquisition support. Speech processing based on the LPC model provides pitch detection, parameter display and editing, formant tracking, speech synthesis, and pattern classification.

Version 6.0 supports data acquisition for hardware from Data Translation, IBM (DACA), Analog Devices, and Metrabyte. In addition, it has built-in conversion facilities for input-output of binary or ASCII data.

The price for Signal Technology's Version 6.0 is $2495.

Reader Service Number 33

C compiler supports Intel 8096 family

Intel Corporation's C-96 compiler supports its 8096 family of 16-bit microcontrollers. The compiler runs on IBM PC XT, AT, or compatible personal computers containing DOS 3.0. The single-pass compiler eliminates intermediate assembly files and reduces operator involvement and compilation time.

Object modules produced by C-96 can be linked with PL/M-96 and ASM-96 object modules. This allows design teams to choose different languages for different software tasks and program in the most efficient and appropriate language for each task.

C-96 generates the "hooks" necessary to allow engineers using the VLSiCE-96 in-circuit emulator and iSBE-96 single-board emulator to take full advantage of the company's symbolic debugging and source code display capability during hardware/software integration.

The C-96 compiler is available for $750 in single quantities. Multiple-copy discounts are available.

Reader Service Number 34

Data entry package supports Harris 9300

Harris Corporation National Accounts Division has announced that the RODE/PC data-entry software package from DPX, Inc., is available for the Harris 9300 network communications system. The software permits high-volume data entry on personal computer workstations networked into the Harris 9300.

RODE/PC is designed for keypunch, key-to-disk, and source data entry applications. Features include data validation at character, field, and screen levels; automatic reformatting; conditional processing; on-line help functions; user exits; and user-defined formatting. The RODE/PC software also supports supervisory operator functions. Host interface is provided by the 9300 RJE, 3270, and asynchronous communication gateways.

Single-user packages of the RODE/PC software for the Harris 9300 are priced at $595 each. The multiuser version, starting at $2125, accommodates up to 16 users.

Reader Service Number 32

VMEbus adapter supports 300M-byte throughput

BBN Advanced Computers has introduced the Butterfly VMEbus Adapter to provide up to 300 Mbytes/s of I/O bandwidth. The adapter allows the Butterfly system to expand to large configurations and maintain high throughput for array processors, graphics systems, and high-speed disk interfaces.

The interconnection network provides all processors with equal access to all memory in the system. Data moves to and from the VMEbus without going through intermediate processor nodes. The adapter operates with a 32-bit address and data VMEbus and consists of two boards driven by a Motorola 68020 microprocessor. One board contains a bus interface and plugs into the backplane; the other board, containing the microprocessor, interfaces to two ports on the Butterfly switch and occupies a slot in the card cage.

The VME Adapter is available from BBN Advanced Computers for $15,000.

Reader Service Number 35

DSP card comes with 6 application packages

The DSP-16 from Ariel Corporation is a data acquistion/signal processor plug-in card for the IBM PC, XT, or AT, which includes a data buffer capable of storing 21 seconds of audio at maximum bandwidth. Bundled with the DSP-16 hardware are six software application packages called the PC Sampler.

The signal acquisition, synthesis, and processing system combines two channels of 50-kHz sample rate, 16-bit-resolution input/output conversion, the data buffer, and the TMS32020 DSP microprocessor. The 5-MIPS throughput of the TMS32020 makes possible complex processing and analysis of the acquired signal in real time, freeing the host computer to set up and control the DSP-16 program, display the processed signal, and store and retrieve data. A separate TMS32020-to-host interface port permits program modification and data transfer on the fly.

The PC Sampler software package includes a program development system and five software application programs: data acquisition, digital audio effects, storage oscilloscope, audio loop editor, and waveform synthesizer. The Program Development System includes driver routine, a TMS32020 assembler, and debug facilities.

List price for quantities one to nine of Ariel Corporation's DSP-16 plug-in card is $2495. OEM quantity discounts are available.

Reader Service Number 36

Package converts raster file to vector data

Microtek Lab is offering its CADmate scanner-to-CAD conversion software for the IBM PC. CADmate converts scanned (raster or bit-mapped) images to vector (line-based) data that is compatible with AutoCAD, VersaCAD, and other PC-based CADD software.

CADmate accepts A-size engineering, architectural, and other drawings from the company's MS-300A scanner at 200 or 300 dpi. It also accepts A- to E-size drawings scanned from the 200-dpi Houston Instrument Scan-CAD plotter accessory.

Microtek Lab prices CADmate at $995.

Reader Service Number 37

AI software repairs PC hard disks

The Disk Technician automated artificial intelligence system prevents, detects, repairs, and recovers hard-disk media failures before data can be lost on IBM PCs and compatibles. According to Prime Solutions Inc., the system takes 60 seconds of daily hands-on operator time to use.

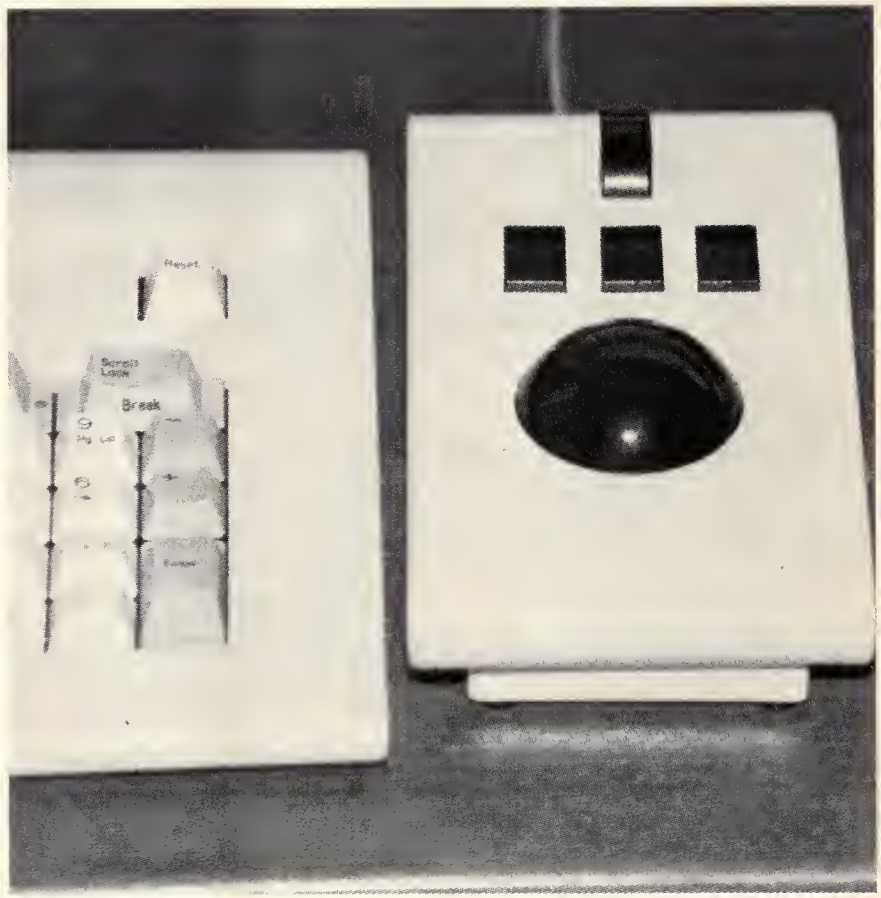
Disk Technician resides on a 5¼-inch diskette and works on hard and floppy disks. The program performs automatic daily, weekly, and monthly hard-disk system testing and repairing of individual bytes on the disk, occupied or not, for reading, writing, track alignment, and magnetic retentivity. All unsafe soft errors are either repaired or blocked.

An early warning detection system in Disk Technician removes unrepairable marginal areas from use. The AI calibration and alignment diskette automatically adjusts Disk Technician to the individual system being checked and features a history and analysis function that "learns" its host system. Pressing P or Enter/Return produces a printed or screen report of test results.

An added feature is SafePark, a memory-resident program that moves the disk head to a safe zone when there has been no activity for seven seconds. In the safe zone data is protected against loss due to power failure or power spikes and fluctuations.

Contact Prime Solutions Inc. for pricing.

Reader Service Number 38



The MicroSpeed FastTrap three-dimensional I/O device has a large tracking surface considered to be very stable in high-resolution graphics applications. The pointing device uses a trackball for x,y motion control and a fingerwheel to control the third, or z, axis. FastTrap has a suggested retail price of $149; delivery is 30 to 60 days ARO.

Reader Service Number 39

Motorola accepts orders for DSP56200 FIR chip

Motorola's Digital Signal Processor Operations has announced the availability of its off-the-shelf DSP56200 finite impulse response filter chip for sampling. Fabricated in 1.5-micrometer CMOS technology, the DSP56200 features the least mean square adaptation algorithm, which is implemented in silicon and eliminates the need for additional programming.

The FIR chip supports applications in general digital filtering and data acquisition systems for DSP products; the company expects it to be particularly useful in communications products requiring adaptive echo cancelling or linear phase digital filtering.

The DSP56200 FIR contains two RAM arrays and a multiply/accumulator. The RAM arrays consist of a 16-bit-by-256 location data RAM and a 24-bit-by-256 location coefficient RAM. A 40-bit product results from the 16-bit-by-24-bit multiplier/accumulator.

An 8-bit data bus and three control lines provide an interface to fast and slow general-purpose processors. The DSP56200 operates in dual-channel FIR filter mode, single-channel FIR filter mode, or single-channel adaptive filter mode. It can be cascaded in both the single-channel FIR and adaptive filter modes. In standby mode the chip retains data and coefficient memory and draws less than 1mA of power.

The 28-pin DIP device is priced at $100. Production quantities are expected to be available fall 1987.

Reader Service Number 40

AT&T announces WE DSP16 chip

AT&T's digital signal processor, the WE DSP16, multiplies and adds instructions simultaneously at a rate of either 75 or 55 nanoseconds, or about 13.3 or 18.2 million instructions per second.

The WE DSP16 is implemented in 1.0-micrometer double CMOS and dissipates less than 0.33 watts of power. The 16-bit IC has an on-board instruction cache that executes a set of up to 15 instructions 127 times with no looping overhead. A parallel pipelined architecture permits different operations to be executed by one DSP16 simultaneously.

Samples of the AT&T WE DSP16 in both speeds are currently available; full production is expected by fall 1987.

Contact AT&T for pricing.

Reader Service Number 41

TI adds to TMS320 DSP family

Texas Instruments has developed its third generation of TMS320 digital signal processors, the TMS320C30. With a computational rate designed to be greater than 33 million floating-point operations per second, the chip can be used in real-time DSP and computation-intensive applications. Its performance level is gained through internal parallelism, large on-chip memories, and concurrent DMA.

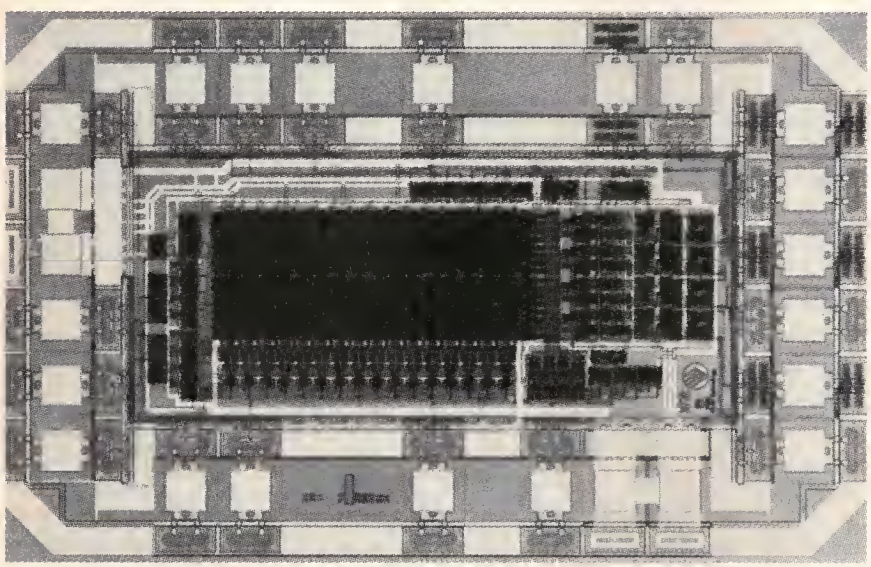
Key features of the TMS320C30 include a 60-ns, single-cycle execution time; two 1K-by-32-bit, single-cycle, dual-access RAM blocks; one 4K-by-32-bit single-cycle, dual-access ROM block; a 64-by-32-bit instruction cache; 32-bit instruction and data words and 24-bit addresses; a 32/40-bit floating-point and integer multiplier; and a 32/40-bit floating-point, integer, and logical ALU. Additionally, the DSP offers eight extended precision registers, two 32-bit address-generator ALUs with eight auxiliary registers, and an on-chip DMA controller for concurrent I/O and CPU operation.

The 1-micrometer CMOS chip is upward-compatible with previous versions of the TMS320 family. Application support and quality development tools available include a full Kernighan and Ritchie C compiler, which supports inline assembly language code.

TI expects sample quantities to be available first quarter 1988 in two versions. A 144-pin microprocessor version is unpriced as yet; an 84-pin microcomputer will most likely be priced from $40 to $50 each in OEM quantities. Production is projected for fourth quarter 1988.

Reader Service Number 43

SRAM performs at 15-ns speeds



Organized as 256 words by 4 bits, the VLSI Technology VT7C122 SRAM is enclosed in 22-pin plastic DIP; it is also available in 25-ns and 35-ns versions.

The VT7C122 1K-bit static RAM from the Application Specific Memory Products Division of VLSI Technology offers access and cycle times of 15 nanoseconds. The CMOS memory chip is designed for applications of cache memories, writeable control stores, and data buffers.

The VT7C122 SRAM is available in sample quantities of 1000 for $10.44 each. Production-level availability is expected by fall 1987.

Reader Service Number 42

Image capture, graphics boards announced

Vutek Systems has introduced the Freeze Frame Image Capture and Super Deluxe EGA boards. Freeze Frame digitizes video images in real time from standard NTSC sources such as a CCTV camera, VCR, or videodisc player and combines the image with text for viewing on a monitor and subsequent storage on a disk. The combined image can be printed on a dot matrix or laser printer. Freeze Frame works with standard EGA or CGA boards in IBM PCs or compatibles.

The Deluxe EGA board allows users to draw 16 colors from a palette of 64 and supports features of the IBM EGA, CGA, PGA, DEGA, and MDA adapters. It also provides keyboard switching when changing from EGA to CGA modes.

Freeze Frame prices start at $1379, and Deluxe EGA retails for $559.

Reader Service Number 44

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 180 Medium 181 Low 182

Product Summary

Marlin H. Mickle
University of Pittsburgh

For more information, circle the appropriate Reader Service Number on the Reader Service Card at the back of the magazine.

Manufacturer	Model	Comments	Rs No.
Chips/Components			
Accutek Microcircuit Corp.	DRAM modules	Family of 6:1-density dynamic RAM modules comes in 18-pin DIP, 22-pin SIP, and 30-pin and 40-pin SIMM configurations, which comply to JEDEC pinout standards. Each device uses 1M, 256K, or 64K chips packaged in LCCs or PLCCs and surface mounted to a multilayer substrate. Prices not stated.	60
Analogic Corp.	ADAM-826 Eurocard	Eurocard-packaged analog-to-digital converter is available in two configurations, the basic A/D converter with 1.5-ms conversion time and another with sample and hold amplifier and speeds of ±0.0015 percent in less than 800 ns for a full 20V step. Price not stated.	61
Integrated Device Technology	IDT75C19/29 DAC converters	CMOS 9-bit, 125-MHz video digital-to-analog converter optimized for artificial vision applications drives a 75-ohm standard load to video levels with 1280 × 1024 resolution. IDT75C19 features ECL-compatible inputs, and the IDT75C29 has TTL-compatible inputs. Packaging includes 24-pin hermetic DIPs; 28-pin LCCs; and 24-pin, 0.300-inch plastic Thindips. $38.50 for commercial-grade Cerdip in 100-up quantities.	62
Boards			
Levco	Prodigy SE	Macintosh SE performance-enhancement board plugs into the SE-Bus slot to change the computer into a portable workstation capable of running applications 100 times faster. The 16-MHz, 32-bit 68020 board features 1M-byte RAM; a built-in, nonvolatile RAM disk; and two on-board expansion ports for adding high-speed memory and peripheral connections. $1995 each.	63
Software			
University of Southern California	ScriptWriter	IBM PC software permits an author to create educational software with tools such as graphics, text, and font editors and the IQ programming language. Users create a set of screens and program the interactions between the user and the system. Requires a 512K XT or AT; sound and animation support is available. Basic system, $40; laser disk monitor support, $20; program library, $20.	64
Peripherals			
Commodore Business Machines	Genlock 1300	Electronic outboard device allows users to superimpose Amiga graphics, animation, stereo sound, and titles over videotaped images generated by video equipment. The 2.5-lb., stand-alone genlocking device synchronizes external video signals for display on a monitor or television set or for videotape recording. $295 each.	65

Calendar

Conferences sponsored or cosponsored by the Computer Society of the IEEE are indicated by the society's logo. Submit information **eight weeks before cover date** to Calendar, *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.

July

ACM SIGGraph 87, July 27-31, Anaheim, California. Contact SIGGraph 87 Conference Management, Smith Bucklin and Associates, Inc., 111 E. Wacker Dr., Suite 600, Chicago, IL 60601; (312) 644-6610.

14th Annual Conference on Computer Graphics and Interactive Techniques (ACM), July 27-31, Anaheim, California. Contact SIGGraph 87 Conference Management, Smith Bucklin and Associates, Inc., Suite 600, Chicago, IL 60601; (312) 644-6610.

August

Eurographics 87 (ACM, IFIP), August 24-28, Amsterdam. Contact Secretariat Eurographics 87, c/o Organisatie Bureau Amsterdam, Europaplein 12, 1078 GZ Amsterdam, The Netherlands; phone 31 (20) 44-08-07.

1987 Annual International Test Conference, August 30-September 4, Washington, DC. Contact Doris Thomas, PO Box 264, Mount Freedom, NJ 07970; (201) 895-5260.

September

Euromicro 87, 13th Symposium on Microprocessing and Microprogramming: Microcomputers—Usage, Methods, and Structures, September 14-17, Portsmouth, England. Contact Chiquita Snippe-Marlisa, p/a TH Twente, gebouw TW/RC, Rm. A227, PO Box 217, 7500 AE Enschede, The Netherlands; phone 31 (53) 33-87-99.

ICCC-ISDN 87, Evolving to Integrated Services Digital Networks in North America, September 15-17, Dallas. Contact Caroline Stites, Bell Atlantic, 1310 N. Court House Rd., tenth floor, Arlington, VA 22201; (703) 974-5453.

Midcon 87 (IEEE), September 15-17, Rosemont, Illinois. Contact Alexes Razevich, Electronic Conventions Management, 8110 Airport Blvd., Los Angeles, CA 90045; (213) 772-2965 or (800) 421-6816.

1987 Design Automation Conference (ASME), September 27-30, Boston. Contact S.S. Rao, School of Mechanical Engineering, Purdue University, West Lafayette, IN 47907; (317) 494-5699.

Fall National Design Engineering Show, Corporate Electronic Publishing Systems, September 29-October 1, New York. Contact Show Manager, Fall National Design Engineering Show, 999 Summer St., Stamford, CT 06905; (203) 964-0000.

October

12th Conference on Local Computer Networks, October 5-7, Minneapolis, Minnesota. Contact Stephane Johnson, Start, Inc., 10301 Toledo Ave. South, Bloomington, MN 55437; (612) 831-2122.

ICCD-87, IEEE International Conference on Computer Design: VLSI in Computers and Processors, October 5-8, Rye Brook, New York. Contact Prathima Agrawal, AT&T Bell Laboratories, 600 Mountain Ave., Rm. 3D-480, Murray Hill, NJ 07974; (201) 582- 6943.

Compsac 87 (Computer Society, IPSJ), October 5-9, Tokyo. Contact Tosiyasu L. Kunii, c/o Business Center for Academic Societies Japan, Yamazaki Bldg. 4F, 2-40-14, Hongo, Bunkyo-ku, Tokyo 113, Japan; phone 81 (3) 817-5831, or Albert K. Hawkes, Sargent & Lundy, Engineering Consultants, 55 E. Monroe, Chicago, IL 60603; (312) 269-3640, or Stephen S. Yau, Northwestern University, Dept. of Electrical Engineering and Computer Science, Evanston, IL 60201; (312) 491-3641.

7th Annual Symposium on Small Computers in the Arts, October 8-11, Philadelphia. Contact Maurice Herlihy, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; (412) 268-2584.

FJCC-87, Fall Joint Computer Conference (Computer Society, ACM), October 25-29, Dallas. Contact Debra Anthony, Texas Instruments, 6500 Chase Oaks Blvd., PO Box 86905, MS 8419, Plano, TX 75086; (214) 575-2151.

FOC/LAN 87, 11th International Fiber Optic Communications and Local Area Networks Exposition, October 26-30, Anaheim, CA. Contact Information Gatekeepers, Inc., 214 Harvard Ave., Boston, MA 02134; (617) 232-3111.

Government Microcircuits Applications Conference, October 27-29, Orlando, Florida. Contact Frank J. Rehm, RADX Griffiss Air Force Base; (315) 330-7781.

November

ICCAD-87, IEEE International Conference on Computer-Aided Design, November 9-12, Santa Clara, California. Contact Basant Chawla, AT&T Bell Laboratories, 1247 S. Cedar Crest Blvd., Allentown, PA 18103; (215) 770-3484.

December

Micro 20, 20th Annual Workshop on Microprogramming (ACM), December 1-4, Colorado Springs, Colorado. Contact Gearold R. Johnson, Center for Computer-Assisted Engineering, Colorado State University, Fort Collins, CO 80523; (303) 491-5543.

ISELDECS-87, International Symposium on Electronic Devices, Circuits, and Systems, December 16-18, Kharagpur, India. Contact N.B. Chakrabarti, Dept. of Electronics and Electrical Comm. Eng., Indian Institute of Technology, Kharagpur 721302, WB, India, or Vishwani Agrawal, (201) 582-4349.

January 1988

Annual IEEE Design Automation Workshop, January 13-15, Apache Junction, Arizona. Contact Walling Cyre, Control Data, HQM 173, Box 1249, Minneapolis, MN 55440; (612) 853-2692.

February 1988

ADEE 88, Automated Design and Engineering for Electronics, February 7-9, New Orleans. Contact ADEE West, Cahners Exposition Group, 1350 East Touhy Ave., PO Box 5060, Des Plaines, IL 60017-5060; (312) 299-9311.

Nepcon West 88, February 23-25, Anaheim, Calif. Contact Jerry Carter, Cahners Exposition Group, 1350 East Touhy Ave., PO Box 5060, Des Plaines, IL 60017-5060; (312) 299-9311.

Advertiser Index

FOR DISPLAY ADVERTISING INFORMATION CONTACT

Southern California and Mountain States: Richard C. Faust Company, 24050 Madison Street, Suite 100, Torrance, CA 90505; (213) 373-9604.
Northern California and Pacific Northwest: Don Farris Company, 161 W. 25th Ave., #102B, San Mateo, CA 94403; (415) 349-2222.
Jack Vance, P.O. Box 3205, Saratoga, CA 95070; (408) 741-0354.
East Coast: Hart Associates, Inc., P.O. Box 339, 42 Lake Blvd., Matawan, NJ 07747; (201) 583-8500.
New England: Arpin Associates, P.O. Box 227, 51 Colchester, Weston, MA 02193; (617) 899-5613.
George Watts, III, 4 Conifer Dr., Wilbraham, MA 01095; (413) 596-4747.
Midwest: Thomas Knorr, Knorr MicroMedia, Inc. 333 North Michigan Ave. Chicago, IL 60601; (312) 726-2633.
Southwest: The House Company, 3000 Weslayan, Suite 345, Houston, TX 77027; (713) 622-2868.
Southeast/Telemarketing: Kay Young and Assoc., 22 Pebblewood, Irvine, CA 92714; (714) 551-4924.
Europe: Heinz J. Görgens, Parkstrasse 8a, D-4054 Nettetal 1-Hinsbeck (F.R.G.); (0 21 53) 8 99 88.

Advertising Director: Dawn Peck, IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-3240, 821-8380.

For production information, conference or classified advertising contact Heidi Rex or Marian Tibayan. IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, (714) 821-8380.

Product Index

READER SERVICE CARD

IEEE MICRO

June 1987 issue
(card void after December 1987)

Name_____

Title_____

Company_____

Address_____

City_____ State___ ZIP_____

Country_____ Phone (____) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **1**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	190
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

READER SERVICE CARD

IEEE MICRO

June 1987 issue
(card void after December 1987)

Name_____

Title_____

Company_____

Address_____

City_____ State___ ZIP_____

Country_____ Phone (____) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **2**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	190
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

SUBSCRIPTION CARD

SUBSCRIBE TO IEEE MICRO

☐ **YES, sign me up!**

If you are a member of the Computer Society or any other IEEE society, pay the member rate of only **$17** for a year's subscription (6 issues). Subscriptions are annualized with membership. For orders submitted March through August, please pay half the given full-year fee for a half-year's subscription.

Society:_____ IEEE membership no:_____

☐ **YES, sign me up!**

If you are a member of ACM, NSPE, ACS, IEE (UK), SCS, IPSJ, IECEJ, or any other technical society, pay the sister society rate of only **$25** for a year's subscription (6 issues).

Society:_____ Mem. no. (if any):_____

_____
Full Signature *Date*

_____
Name

_____
Street

_____
City

_____
State/Country *ZIP/Postal Code*

☐ Payment enclosed

☐ Charge to ☐ Visa ☐ MasterCard ☐ AmEx

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐
Charge Card Number

Mo. Yr.
☐☐ ☐☐
Exp. Date

M687

Charge orders also taken by phone:
(714) 821-8380 8:00 a.m. to 5:00 p.m. Pacific time
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720

Editorial comments

I liked: _____

I disliked: _____

I would like to see: _____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91605
USA

Editorial comments

I liked: _____

I disliked: _____

I would like to see: _____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91605
USA

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 38 LOS ALAMITOS, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Computer Society of the IEEE
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-9970

THE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Purpose

The Computer Society strives to advance the theory and practice of computer science and engineering. It promotes the exchange of technical information among its 90,000 members around the world, and provides a wide range of services which are available to both members and non-members.

Membership

Members receive the highly acclaimed monthly magazine *Computer*, discounts on all society publications, discounts to attend conferences, and opportunities to serve in various capacities. Membership is open to members, associate members, and student members of the IEEE, and to non-IEEE members who qualify as affiliate members of the Computer Society.

Publications

Periodicals. The society publishes six magazines (*Computer, IEEE Computer Graphics and Applications, IEEE Design & Test of Computers, IEEE Expert, IEEE Micro, IEEE Software*) and three research publications (*IEEE Transactions on Computers, IEEE Transactions on Pattern Analysis and Machine Intelligence, IEEE Transactions on Software Engineering*).

Conference Proceedings, Tutorial Texts, Standards Documents. The society publishes more than 100 new titles every year.

Computer. Received by all society members, *Computer* is an authoritative, easy-to-read monthly magazine containing tutorial, survey, and in-depth technical articles across the breath of the computer field. Departments contain general and Computer Society news, conference coverage and calendar, interviews, new product and book reviews, etc.

All publications are available to members, nonmembers, libraries, and organizations.

Activities

Chapters. Over 100 regular and over 100 student chapters around the world provide the opportunity to interact with local colleagues, hear experts discuss technical issues, and serve the local professional community.

Technical Committees. Over 30 TCs provide the opportunity to interact with peers in technical specialty areas, receive newsletters, conduct conferences, tutorials, etc.

Standards Working Groups. Draft standards are written by over 60 SWGs in all areas of computer technology; after approval via vote, they become IEEE standards used throughout the industrial world.

Conferences/Educational Activities. The society holds about 100 conferences each year around the world and sponsors many educational activities, including computing sciences accreditation.

European Office

This office processes Computer Society membership applications and handles publication orders. Payments are accepted by cheques in Belgian francs, pounds sterling, German marks, Swiss francs, or US dollars, or by American Express, Eurocard, MasterCard, or Visa credit cards.

Ombudsman

Members experiencing problems — late magazines, membership status problems, no answer to complaints — may write to the ombudsman at the Publications Office.

Information

Use the Reader Service Card to obtain the following material:

- Membership information and application (RS #202)
- Publications catalog (proceedings, tutorials, standards) (RS #201)
- Periodicals subscription application/information for individuals (members, sister-society members, others) (RS #200)
- Periodicals subscription application/information for organizations (libraries, companies, etc.) (RS #199)
- List of awards and award nomination forms (RS #198)
- Technical committee list and membership application (RS #197)
- Directory of officers, board members, committee chairs, representatives, staff, chapters, standards working groups, etc. (RS #196)

IN YOUR FUTURE...

THE NEW! FREE! 1987 OMEGA HANDBOOKS AND ENCYCLOPEDIAS TELL YOU ALL!

• TEMPERATURE • PRESSURE • FLOW • pH • STRAIN • DATA LOGGING • THERMISTORS • RTD's • READOUT DEVICES

IN A HURRY FOR YOUR HANDBOOKS? DIAL

OMEGA®
ENGINEERING, INC.
(203) 359-RUSH

One Omega Drive, Box 4047, Stamford, CT 06907
Telex 996404 Cable OMEGA FAX (203) 359-7700

CIRCLE READER SERVICE NUMBER
OR SEND BUSINESS CARD
TO RECEIVE QUALIFICATION FORM.

Reader Service Number 1